NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SPEEDING UP A PATH-BASED POLICY LANGUAGE COMPILER

by

Ahmet Guven

March 2003

Thesis Advisor:	Geoffrey Xie
Thesis Co-Advisor:	Neil Rowe

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY	**2. REPORT DATE** March 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Speeding Up A Path-Based Policy Language Compiler			**5. FUNDING NUMBERS**
6. AUTHOR(S) Ahmet Guven			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT Policy based network management has an increasing importance depending on the increasing importance of distributed large networks and the growing number of services that run on them. Policy languages, which enable users define policies in a formal language, are one of the main tools of policy management. Even though there are policy languages like PFDL or RPSL, none of them has the capability of a robust conflict detection and resolution focused on policy.

A new Policy Language, Path-based Policy Language (PPL), has been developed recently. It encompasses as many of the features addressed in the other policy languages as possible, as well as providing means for testing policies for consistency and defining both static and dynamic policies. The most important, PPL provides the ability to detect and resolve conflicts between by translating policy rules into formal logic statement and checking them with a Prolog program.

Even though in theory PPL seems to be a very high performance policy language, its current compiler has a performance bottleneck. In some cases the PPL compiler can not finish compilation and runs forever without returning any conflict results. This thesis focuses on the PPL compiler's performance bottleneck and introduces solutions speeding up the PPL compiler. The new PPL compiler achieves a reasonable compilation time for any configuration file for a network with 100 nodes while maintaining its ability to detect and resolve policy conflicts.

14. SUBJECT TERMS Policy, Policy Language, Path-Based Policy Language, Compiler, Conflict, Conflict Detection, Conflict Resolution, Prolog , Articulation Point, Biconnected Components, Depth Limited Bidirectional Search.			**15. NUMBER OF PAGES** 167
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

SPEEDING UP PATH-BASED POLICY LANGUAGE COMPILER

Ahmet Guven
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2003

Author: Ahmet Guven

Approved by: Geoffrey Xie
Thesis Advisor

Neil Rowe
Thesis Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Policy based network management has an increasing importance depending on the increasing importance of distributed large networks and the growing number of services that run on them. Policy languages, which enable users define policies in a formal language, are one of the main tools of policy management. Even though there are policy languages like PFDL or RPSL, none of them has the capability of a robust conflict detection and resolution focused on policy.

A new Policy Language, Path-based Policy Language (PPL) [10], has been developed recently. It encompasses as many of the features addressed in the other policy languages as possible, as well as providing means for testing policies for consistency and defining both static and dynamic policies. PPL's path-based approach enables establishing policies that will be based on path, like Integrated Services, as well as non-path based policies, which are more suited for Differentiated Services. The most important, PPL provides the ability to detect and resolve conflicts between by translating policy rules into formal logic statement and checking them with a Prolog program.

Even though in theory PPL seems to be a very high performance policy language, its current compiler has a performance bottleneck. In some cases the PPL compiler can not finish compilation and runs forever without returning any conflict results. This thesis focuses on the PPL compiler's performance bottleneck and introduces solutions speeding up the PPL compiler. The new PPL compiler achieves a reasonable compilation time for any configuration file for a network with up to several hundred nodes while maintaining its ability to detect and resolve policy conflicts.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This thesis is dedicated to my parents, Ali Guven and Hacer Guven, for their lifetime support and encouragement for my education. My love for them is the reason I have succeeded.

My sincerest gratitude goes to Professor Neil Rowe. He was there whenever I needed help with his great knowledge about any area in computer science. He helped me solve the hardest Prolog programming problems in this work. I appreciate not only what he taught me, but also the elements of being a professional in this field.

I would also like to thank Professor Geoffrey Xie for his patience, understanding and motivation, which was very important in finishing this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The increasing importance of distributed large networks and the growing number of services that run on them increase the need for effective network management. To fill up a need for dynamic change in the behaviors of distributed management agents at run-time, policy-based management is emerging as a sound mechanism to assist this. To support a large distributed environment, it is necessary to reason about policies and make policy decisions at run-time.

Network policies are "traffic regulations" for the networks, which make up the Internet. These are necessary for managing the flow of data, for access control to the network, and for managing the network in achieving other types of quality of service goals. Network policy allows administrators to manage network elements to provide service to a set of clients. Policy languages are the tools we use to declare our policies in a formal way. Currently popular policy languages like the Policy Framework Definition Language (PFDL) or the Routing Policy Specification Language (RPSL) do not have the capability of a robust conflict detection and resolution focused on policy.

A new Policy language, Path-based Policy Language (PPL), which allows testing policies for consistency and defining both static and dynamic policies, has been developed for Server-Agent Active Network Management (SAAM) project [29]. PPL enables us to establish policies based on path like integrated services as well as non-path based policies like those for differentiated services. PPL can detect and resolve conflicts between policies by translating policy rules into formal logic.

Although in principle PPL should be efficient, in its current implementation there are performance bottlenecks. Compilation time of a PPL configuration file depends on the number of edges and nodes in the network, the number of policies to be applied, and the way policy paths represented in the configuration file. Some seemingly easy cases take a long time to run.

B. THESIS OBJECTIVES

The main objective is this thesis is to analyze the PPL compiler's performance bottlenecks and resolve them. The target is a reasonable compilation time for any configuration file for a network of several hundred nodes without compromising the compiler's ability to detect and resolve policy conflicts.

C. THESIS ORGANIZATION

This thesis is organized in seven chapters. To understand the problem that the thesis tries to address, it is essential to have a basic understanding of policy-based networking, Prolog, and PPL. To this end, the next two chapters present background information about these related topics. Chapter IV identifies the key bottlenecks in the PPL compiler and explains why they are bottlenecks. Chapter V describes the solutions developed by this thesis. Chapter VI reports results from tests using the developed solutions. Chapter VII presents conclusions drawn from the research and suggests areas for future work.

II. BACKGROUND

A. POLICIES

A policy is "about the constraints and preferences on the state, or on the state transition of a system [1]". Each policy rule is comprised of a set of conditions and a corresponding set of actions. The conditions define when the policy rule is applicable. Once a policy rule is so activated, one or more actions contained by that policy rule may then be executed. "These actions are associated with either meeting or not meeting the set of conditions specified in the policy rule" [2]. In short, when a set of associated conditions are met, a policy specifies what actions must be taken.

Policies must be clear enough so that potential conflicts among them can be detected and resolved. One way to resolve conflicts is imposing a priority and/or order on both the satisfaction of policy conditions as well as the execution of policy actions.

For networks, a policy constrains communication. Network policy defines the association between clients using network resources and those network elements that provide those resources. A *client* in this case refers to users as well as applications and services. Network policies allow administrators to handle network elements to provide service to a set of clients. They guide the administrators to achieve overall objectives on the network. For example, the overall objective of a manager for a system could be "to ensure 95% system up time for the users".

Policy benefits include:

- Provide access control for network resources
- Ensure applications critical to enterprise operations are accomplished
- Deliver tiered bandwidth and differentiated services to each customer according to their needs
- Manage the overall flow of traffic through internal and external networks

In writing policies, the purpose should be to reach the overall objectives. There are resources, clients (users), and requests made by users. But unfortunately sometimes

our resources are not enough to reply to all requests, or sometimes we do not want some users to use our resources. These limitations are reasons that we need policies.

Network policies are usually grouped into two general categories. In [3] and [4] these are *obligation* and *authorization* policies; [5] refers them as *imperatival* and *authority* policies; and [6] and [7] refer to *motivation* or *authorization* policies. The distinction is whether a policy entails some action or concerns the granting of privileges [2].

B. POLICIES, PRACTICES, AND PROCEDURES

Policies are different from practices and procedures. Policies are more general and abstract than practices, which in turn are more general than procedures [8]. Sample policies could be "Every student will have access to e-mail and calendaring applications," and "Permit research class traffic to and from the networks NPS and NSF." Sample practices could be "New students will be assigned e-mail and calendaring accounts on their first school day," and "Only members of the Faculty_Management group will be permitted to look up confidential student data." A sample procedure could be: "Student mail accounts are named according to the following system: Firstname_Lastname. Duplicate first and last names are resolved by having the students with the least seniority insert their middle name in this way: Firstname_Middlename_Lastname. It is the user's responsibility to archive messages." Another example of a procedure could be: "Members of Faculty will receive read-access rights to tables X, Y, and Z in the Student database. They will use a six-digit PIN to log in."

Policies are generally expressed in ordinary business language; they don't address implementation methods. Practices on the other hand address implementation methods, but not at the level of specific products, data structures, and keystrokes; these specifics are covered in procedures. The advantage of having three specification levels is that changing particular products or vendors should not generally require changing practices. Additionally, practices that do a better job of carrying out policies can be adopted without high-level policy debates. Wise organizations encourage broad participation when formulating policies. Widespread understanding of why policies are the way they are be

the case if policies were mandated by a small group of executives or the information technology staff. Moving from practices to procedures is a technical (and budgetary) matter that should be relatively free of organizational politics.

Different organizations have various levels of commitment to developing and maintaining documents related to policies, practices, and procedures. Large, distributed enterprises with numerous resources will typically have a greater need than small organizations for documenting uniform policies, practices, and procedures. For educational institutions, spelling out user responsibilities and prescribing consequences for abusive activities might have a high priority.

C. POLICY-BASED NETWORKING

"Policy-based networking is a set of automated rules to control congestion. These rules govern which users or applications can use specified network bandwidth at any given time" [9]. Policy-based networking helps manage user and application priorities, efficiency, and security rights based on organizational policies.

With the convergence of data, telephone, and video traffic in the same network, companies will be challenged to manage traffic so that one kind of service does not preempt another kind. Using policy statements, network administrators can specify which kinds of service to give priority to at what times of day on what parts of their Internet Protocol (IP)-based network. What governed by this kind of management is often known as Quality of Service (QoS) and is controlled using policy-based networking software. A policy statement could be as natural as: "Provide the fastest forwarding for all voice traffic to NPS between 9 am and 3pm."

1. Definitions

Policy is comprised of three functions: decision-making, enforcement and policing. *Decision-making* is the process of deciding to what action take for the desired state. *Enforcement* means the implementation of the decision. *Policing* is the examination of the network to see whether the policy requirements are being satisfied or not. A network policy defines some regulations limiting relationships between clients

that desire service and the network elements that provide those services. Here, clients refer to users as well as applications and services.

The policy repository is one of three important entities of the model. The other entities are the Policy Enforcement Points (PEP) and Policy Decision Point (PDP). The PEP is a component of a network node where the policy decisions are actually enforced. It may be a router, switch or hub. PEP is not able to make decisions itself. When the PEP requires a policy decision about a new flow of traffic, or authentication for example, the PEP will send a request to a PDP. The PDP is the entity in the network where policy decisions are made. Any PEP that encounters an event requiring a policy based decision first asks a PDP how to handle this request. PDP may make one ore more policy decisions related to this request. Figure 1 shows an illustration of common policy-based network management architecture proposed by the Internet Engineering Task Force (IETF) [2]. Each device does not necessarily have to be separate as shown on the figure nor is it required to use the same protocol. The Lightweight Directory Access Protocol (LDAP) enables the communication to and from the policy repository. The Common Open Policy Service protocol (COPS) [19] can be used to communicate information between the PDP and PEP.

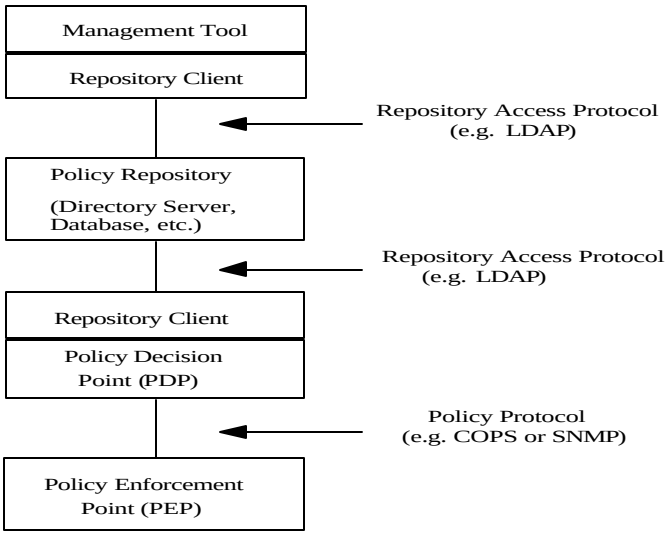

Figure 1. Generic Policy Based Architecture. [After 2]

D. PROLOG

1. Introduction

Prolog is a computer language that supports logic programming, i.e., programming based on first order predicate logic [11]. Like other logic programs, Prolog code is easy to create and enables machines to explain their results and actions. The roots of the Prolog language go back to the branch of logic called *predicate calculus*, which mathematicians and logicians use to make assertions about the world. Then they use these assertions to prove theorems. For example, by asserting that "Socrates is a man" and that "All men are mortal", it is possible to prove that "Socrates is mortal" [24]. The Prolog operation or "search strategy" is based on these kinds of logical assertions and proofs.

Logic programming is programming by description. Thus, Prolog is classified as a *declarative language*. The programmer describes the application area and lets the program choose specific operations [23]. In other words, the programmer states the facts and rules which are relevant to the solution of the problem, and this information is then used as a program to solve the problem without the need for more advice on how to handle the information [12]. *Rules* are conditional statements, which tell Prolog to prove whether something is true [13]. *Facts* are rules with no "if" parts. They are analogous to logical assertions. *Facts* and *rules* together establish the Prolog knowledge base. Users can make *queries* to the knowledge base by using a Prolog interpreter. When a query is made, the interpreter searches the knowledge base and tries to find a matching fact. The order of the rules and facts in the database is the order they are tried when more than one can apply to a situation [26]. An answer for a query may be simply "yes" or "no" or a *variable binding*.

The method Prolog uses for reasoning is called *backward chaining*. It involves working backwards, searching for justifications for a conclusion. A Prolog search starts with a hypothesis and then reasons backwards in the inference network to confirm or refute the hypothesis. Backward chaining helps to draw the conclusions only that are relevant to the goal. Given a goal state to prove, the system will first check if the goal

matches the initial facts given. If it does, then that goal succeeds. Otherwise, the system will look for rules whose conclusions match the goal. One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting them as new goals to prove. Backward chaining does not need to update a working memory. Instead it needs to keep track of what goals it needs to prove to prove its main hypothesis. A disadvantage of backward chaining is that the user has to state all the relevant information as facts in advance, so there is a danger of stating too much or too little**.** Figure 2 shows a backward chaining process.

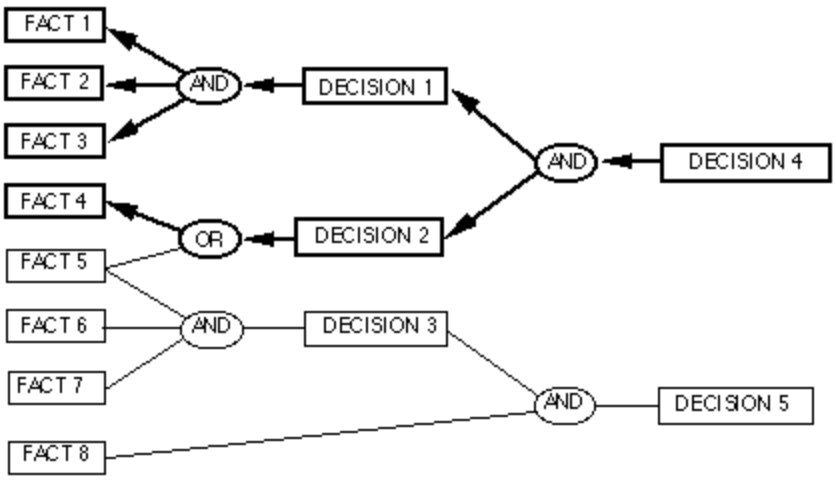

Figure 2. A backward chaining process [From 27].

2. Important Prolog Features

Though Prolog can be a difficult language to master, Prolog programs are usually short when compared to their procedural equivalents. Prolog has other advantages too. Flexible *variable binding* is one of the important features of Prolog. In Prolog, unlike in imperative programming, variables can remain uninitialized indefinitely [25]. The possible values of variables are *terms*, which may be *atoms* (names of individuals and predicates), integers, structures, or clauses. Variables can get bound in query matching and may be returned as a possible answer to the query.

While the interpreter searches for an answer to a query, it checks defined rules. If a rule matches a query expression, all variable bindings in the query expression are

copied to the corresponding variables in the conclusion of the rule. If the rule is to succeed, none of the bindings should conflict with a constant in the conclusion of the rule. This is "head matching" with "call-by-value" [26]. When a rule succeeds, all bindings (besides head matching) that it made of its conclusion's variables are copied to the corresponding variables in the calling expression [26].

The other important feature of Prolog is automatic *backtracking*. There may be several rules unifying with a query. If a rule fails, Prolog automatically backs up and tries another rule to return an answer for the query. Prolog uses the same backup process for the expressions in the rules. If an expression fails, Prolog interpreter returns to the previous expression (if any) and tries to find a new way to satisfy it [26].

3. An Example of How Prolog Works

A simple example helps understanding better how Prolog works. Consider two pieces of information that have been inputted into the Prolog knowledge base:

1- For any X, If X is in California, then X in United States.

2-Montery is in California.

Assume user wants to learn whether Monterey is in United States and ask this question to the computer:

Is Monterey In United States?

By looking at its knowledge base, the Prolog interpreter can:

1-Prove that X is in California, in case it to be in United States.

2-Unify X with Monterey and see whether it is in California.

After these steps, the answer is "`true`", because the knowledge database indicates that `monterey(X)` is in California, so `monterey(X)` is in United States. In the Prolog notation, the knowledge base would be:

```
1- in_united_states(X):- in_california(X).
2- in_california(monterey).
```

Here, statement-1 is a rule and statement-2 is a fact; `in_united_states` and `in_california` are predicates which are type assertions for their arguments. A predicate can take any fixed number of arguments. This number of arguments is the *arity* of the predicate. `monterey` is an *atom*, because it is a constant name. Statement-1 is also a *clause*, which constructs a *structure* with statement-2. One important point to know is that only variables in Prolog start with a capital. *Terms* are the single data structure in logic programming. The definition is inductive. Constants, variables or structures are *terms.*

a. Unification and Variable Instantiation

The first step in solving any query is to match or *unify* the query with a fact or the left hand side (the *head*) of a rule [15]. Unification can assign a value to a variable in order to achieve a match; which is referred as *instantiating* the variable.

Consider the database with five clauses below. The user is trying to learn whether Monterey is in North America.

```
1- in_north_america(X):- in_united_states(X).
2- in_united_states(X):- in_texas(X)
3- in_united_states(X):- in_california(X).
4- in_texas(austin).
5- in_california(monterey).
```

```
Goal        : in_north_america(monterey).
Clause      : in_north_america(X):- in_united_states(X).
Instantiation : X = monterey
New Goal    : in_united_states(monterey).
```

The new query can be unified with the third clause as follows:

```
Goal        : in_united_states(monterey).
```

Clause : `in_united_states(X):- in_california(X).`

Instantiation : `X = monterey`

New Goal : `in_california(monterey).`

This query matches the clause-5. Because there is no "if" part in clause-5, there is no need for more queries; the answer will be "`Yes`" for the main query.

b. Backtracking

There are two rules unifying with the query above. Prolog could have unified clause-4 first with the query. Then the new goal would be:

```
in_texas(monterey).
```

which would *fail*. As in other modern programming languages, such an erroneous call does not abnormally terminate the program's execution, but activates an error handling mechanism. In contrast to other languages, however, the error is not necessarily performed by an active procedure present in the activation stack [16]. Prolog uses a more general method and takes into account even those procedures which returned to the user after successful termination.

Prolog attempts to satisfy the goal in a left to right progression; which is called "goal-directed depth first search" [17]. Each goal to be satisfied generates a sequence of searches for Prolog to perform that can be visually presented as a tree structure. The search tree of the knowledge base for the goal above is shown below in Figure 3. Because the knowledge base has no fact `in_texas(monterey)`, the query fails. But because there is another clause which can be unified with our goal, Prolog *backtracks* and this time tries to find the answer by unifying the goal with the other clause. This attempt succeeds and the Prolog exits the query with the answer "`Yes`".

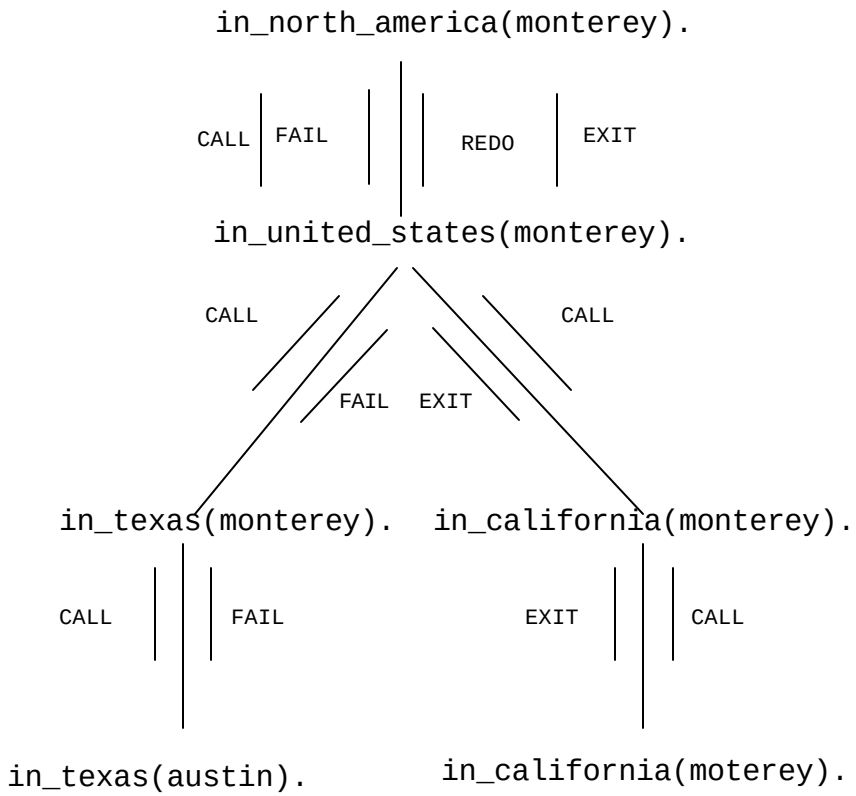

Figure 3. A sample Prolog search tree.

III. THE PATH-BASED POLICY LANGUAGE (PPL)

A. INTRODUCTION

There is a need for tools enabling users to apply policies over networks. One of the languages developed for this purpose is Path-based Policy Language (PPL) [10]. The purpose for creating it was to solve and/or alleviate many of the deficiencies of the previous policy languages. This new language has "the ability to represent network policies unambiguously, providing support to heterogeneous networks for which the networks are controlled using explicit policies." [10]. PPL supports both path and non-path based traffic flows. It also has the ability to support conflict detection and resolution with the use of formal logic.

B. GOALS

The main goal of PPL is to support policies applicable to differentiated service, integrated service as well as multiprotocol label switching models proposed by The Internet Engineering Task Force (IETF). The PPL's goals [10] are listed in Table 1.

	PPL GOALS
1	Create a path-based representation of policies flexible enough to support both path and non-path based traffic flows.
2	Represent network policies in an unambiguous way
3	Be abstract enough to cross device and manufacturer boundaries
4	Detect and resolve conflicts between polices
5	Support the dynamic aspect of networks

Table 1. Summary of the goals of PPL [From 10].

One of the outstanding abilities of PPL is supporting policies based on dynamic factors such as packet delay and packet loss rate, along with static policies on attributes such as traffic class or network address. Because the state of a network is rarely constant, a good policy language should be able to represent network policies based on dynamic factors as well as constant ones. To support a dynamic network, PPL provides the ability to react to dynamic conditions of the network. The *messages* that can be defined by policy maker provide feedback to the system about the current state of the network for a reaction.

C. PPL POLICY FORMAT

A PPL rule consists of six elements which must be ordered for the format shown in Figure 4.

PolicyID userID {paths} {target} {conditions} {action_items}

policyID - unique policy identification token

userID - user ID of policy creator

paths - network paths that the policy affects. They must be defined by the users in formal language.

target - user defined target class of network traffic, like *data, video, voice* etc. For more than one traffic class, items are disjunctive.

conditions -any global conditions like a specific day, time or dynamic conditions like a bandwidth value. Dynamic conditions are controlled by *messages*. With more than one defined condition, items are conjunctive.

action_items - includes setting parameters (e.g. priority), deny or permit, etc.

Figure 4. Summary of PPL format [From 10]

For scalability in large networks, PPL has a *wildcard* ("*") character. It can be used in policy definitions to represent all paths, all paths between two spesific nodes, all

target traffic classes, or all global conditions that can be used to create policies. A few examples will make clear how to create policies in PPL.

Example 1: Policy 1 net_manager {<5,7,9>} {traffic_class == {data}} {*} {priority := 3};

policyID : Policy 1

userID : net_manager

paths : <5,7,9>

target : {traffic_class == {data}}

conditions : {*}- All

action_items: {priority := 3}.

This rule states that the path starting at node 5, traversing to node 7, and ending at node 9 will provide priority level 3 for data traffic at any conditions. Here the use "*" character can be seen.

Example 2: Policy4 xie{*}{traffic_class = {data}}{allotted_bw() == 45MBPS, loss_rate () <25%} {allotted_bw := 35MBPS};

policyID : Policy 4

userID : xie

paths : {*}-All

target : {traffic_class == { data }}

conditions : {allotted_bw() == 45MBPS, loss_rate () <25%}

action_items: {allotted_bw := 35MBPS}

This policy shows the ability of supporting dynamic policies by the user created *messages*. Data traffic is provided with an allotted bandwidth of 45 Mb/s, but when the network loss rate is less than 25%, the allotted bandwidth will be lowered to 40 Mb/s.

D. PPL DETAILS

1. How PPL Supports Integrated Services

As mentioned one of the purposes of PPL is to support both integrated and differentiated services. The "path based" structure of PPL enables users support integrated services, which provide QoS assurances on a per-flow basis. A PPL policy may allocate and reserve enough resources for any specific flow of packets. Any path which cannot support that policy will be rejected.

In PPL the paths which are affected by that policy must be specified. The users have the flexibility of including many paths for that policy as long as they separate them by a "," character. The example below shows a policy that the user wants to apply to four different paths. The paths are also defined and named by the user.

Policy1 guven {path1, path2,path5,path12}{*}{*}{deny}

It was mentioned that the paths are defined by the user. This is also a very easy process with PPL "define path" statements. For example "path1" in the example might have been defined by:

define path path1{NPS, SPAWAR,DARPA}

This says that the user wants to include nodes NPS, SPAWAR and DARPA in a path named path1.

The nodes in the network must be defined by a "define node" sentence. For a higher granularity of QoS application, PPL enables use apply policies even to specific nodes. The relations among those nodes (links) are identified with a "define link" sentence.

define node NPS, DARPA, SPAWAR

define link link1<NPS, DARPA>

define link link2<DARPA, SPAWAR>

There may be policies which user wants to apply the whole network or to all possible paths in the network, for which the user can use the wildcard ("*") character.

The wildcard character can also be used for the other elements in the PPL format like *target* or *conditions*. The sample policy below denies all traffic under all conditions in the whole network. This is a *static* policy which does not depend on dynamic aspects like 'delay' or 'loss rate'.

```
Policy2 net_manager {*} {*} {*}{DENY}
```

For integrated services based on certain conditions through a path PPL provides a *conditions* argument in a policy statement. To execute that policy, all the conditions listed in a policy rule must be met. The following examples show usage

```
Policy3 net_manager {path1} {*}}{time <=1700} {DENY};
```

```
Policy4 net_manager {*}{traffic_class == {video}}{jitter() <2 msec, delay() < 4 msec}{PERMIT};
```

Policy3 can be applied only before 5.00 pm. Policy4 permits "video" traffic only when jitter is less than 2 msec and delay is less than 4 msec.

As can be seen in the examples, a user defines at the end of policy statement an *action* that must be taken. The user can choose one of DENY/PERMIT or a dynamic reaction as an *action_item*. The following example shows how to using a dynamic *action_item* in a policy statement. Policy4 gives *data* class of traffic first priority after 5.00 p.m.

```
Policy4 guven {Path1}{ traffic_class == {video}} {time>1700} {priority := 1}
```

2. How PPL Supports Differentiated Services

The logic behind the differentiated services is offering different services to differently marked packets by creating several packet classes. This concept allows, for example, giving different priorities to packets that belong to students and those that belong to faculty members, or to packets of data and to packets of voice. The user should define the required classes of traffic and user classes for those traffic classes that may be used.

The priorities that may be given for any traffic class should also be defined by the user as a different class. The priority of that class of traffic is the main factor for a node when it needs to make a decision between two types of traffic classes. The examples [10] below show how a user can define classes of traffic, users and priorities:

```
define class traffic_class {data, video, voice};

define class traffic_priority {high, med, low};

define class user {faculty, student, staff};
```

To use this, the fourth element in PPL format is the *target* of the policy:

```
Policy1 guven{Path1} {traffic_class == {video}, {traffic_priority == {high}}{*}
{PERMIT};
```

This policy permits on Path1 only *video* traffic that has *high* priority. When multiple classes of traffic are presented in a policy statement as a comma separated list, the classes are logical disjuncts [10].

3. How PPL Supports Dynamic Policies

PPL enables users to define policies not only for static network conditions, but also dynamic aspects like "delay" or "loss rate". Because those values can change anytime in the network, we need a feedback mechanism from the current state of network. User defined *messages* represent information update about some measurable attribute of a network. For example a *message* identified as *delay()* might provide the time required to travel from point A to point B in a network in milliseconds [10].

To use a *message* element, the user must associate the message with a path. The example below shows this. The user associates a *loss_rate()* message with the Path1 and also assigns a bandwidth to Path1 by using *define path_param* statement. This policy denies student traffic when the loss rate on the Path1 is over 40%.

```
define path_param Path1 {BW := 100 MBPS, loss_rate()};

Policy3 net_manager {Path1}{traffic_class == {student}} {loss_rate() > 40%}
{DENY};
```

E. POLICY CONFLICT DETECTION AND RESOLUTION

One of the major advantages of PPL is its ability to detect and resolve policy conflicts unambiguously. The previous languages were not as powerful as PPL in this aspect. The conflict detection and resolution process starts at the PPL compiler by parsing written policies. While parsing, the compiler verifies that policies are written in correct format and grammar. Then a formal logic representation of those policies is created [10]. The created file represents the network (all nodes, links, policies, etc.) in logic statements.

The reason for creating a file, which represents the network in logic statements, is that Prolog language is being used for detection and resolution of conflicts in the policies. As explained before, Prolog is a language for logical reasoning, which has automatic backtracking and flexible variable binding. Because the items to be checked are a set of policy statements defined on a network, these features enable Prolog to be good for a conflict detection and resolution process. We describe "what causes conflict" (rules) by Prolog rules and have the interpreter search the policy database for conflicts.

Figure 5 shows the PPL file compilation process that is proposed in [10]. Compilation starts with the acceptance of a PPL configuration file containing network construction information and network policies. The result of parsing this file is a representation of the entered info in logic statements. This is the starting point of a three-stage process of Prolog conflict detection and resolution. Although the three stages could have been accomplished in just one stage, multiple stages provide an incremental means to verify correctness [10]. The three Prolog stages will be overviewed in Chapter V.

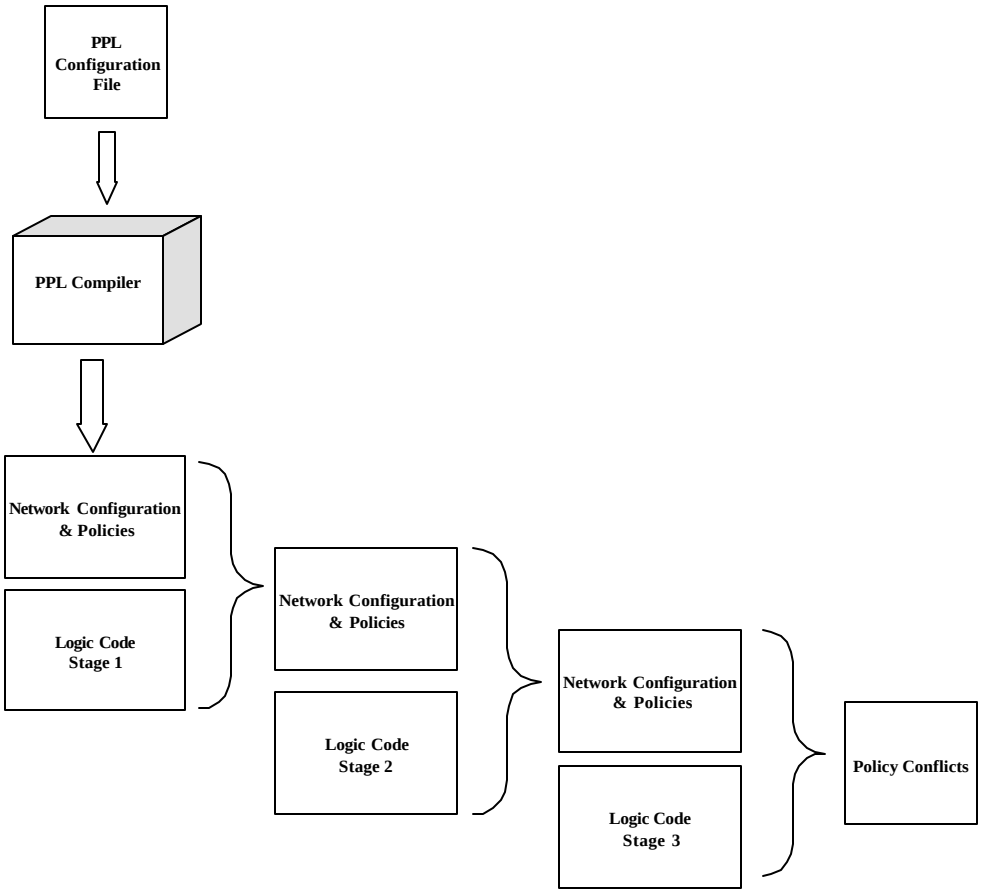

Figure 5. Overall process flow [From 10].

The output file after the Prolog stage 3 lists the resolved and unresolved conflicts between policies in an ASCII format. If two policies are created by users having a different level of priorities, conflicts are resolved in favor of the higher-priority policy creator. The operator can see both resolved and unresolved conflicts by checking the output file.

1. Detecting Conflicts

Because PPL is based on *paths*, policies effective on the same paths are compared for conflicts. If there is no overlap between two paths, no need exists for comparing policies defined on them for conflicts.

PPL uses the general definition of conflict for two network policies to describe how conflicts occur:

Given a set of policies *P*, a conflict exists if for any 2 policies *r, s* $\hat{I}$ *P*, all of the following hold:

1. Physical paths of *r* and *s* overlap on at least one path segment;
2. Time and network conditions specified in the *conditions* elements of *r* and *s* overlap;
3. There is a *target a* which is permitted in *r*, but denied in *s* (or conversely).

The PPL conflict detection process in [10] is based on the definition above. First, all policies that contain overlapping *paths* (physical) are found. After this step, the number of overlapping paths to examine is decreased by picking only the ones that have overlapping *conditions* (timing, network conditions, etc.). By comparing *target* & *action_item* of those polices it can be determined whether those policies conflict or not.

Conditions are the rules restricting flow of traffic on defined paths. These conditions may limit the traffic hours or may only permit traffic when the bandwidth has a previously defined value (dynamic).

If two paths have overlapping points, and they have policies defined for the same conditions (for example both policies allows traffic only on weekends), their *target* and *action_items* must be checked to detect the conflict. Target refers to the class of traffic for which the policy works. If those overlapping policies do not have the same target traffic class, there is no conflict; otherwise, the *action item* of policies must be identified. If one of the policies *denies* and the other *permits* the same target class of traffic, it is considered as a conflict.

2. Some Conflict Samples

Conflicts in PPL occur when one of the policies permits a specific kind (target) of traffic, even though another policy does not permit that kind of traffic under the same

conditions on some common links. Conditions of a policy define the time when the policy is active (like certain days of week, or certain times in a day) or characterize target traffic based on packet header information. If there are more than one condition item, they are conjunctive. It is clear that a policy permitting data traffic on defined path1 for all days of week will conflict with a policy denying data traffic on path1 even one day of the week. The samples below illustrate some possible policy conflicts.

```
Policy1 Net_Manager {NPS_DARPA}{traffic_class == {video}}{time >= 1100}
        {deny};
```

Policy1 is a sample policy created by network manager. It denies video traffic after 11.00 a.m on the NPS-DARPA link. The time representation in PPL is based on 24 hour system.

```
Policy2 Net_Manager {NPS_DARPA}{traffic_class == {video}}{time >= 1100}
        {permit};
```

Policy2 permits video traffic on the same link after 11.00 a.m. Policy1 denies what Policy2 allows, so they conflict with each other.

```
Policy3 Net_Manager {NPS_DARPA} {*}{time >= 1100}{permit};
```

This policy permits any kind of traffic after 11.00 a.m. But Policy1 denies video class traffic at the same time period, and conflicts with it.

```
Policy4 Net_Manager {*}{traffic_class == {video}}{time >= 1100}
        {permit};
```

This policy allows video traffic after 11.00 a.m. on the whole network and also conflicts with Policy1.

```
Policy5 Net_Manager {*}{*}{*}{permit};
```

This policy permits any kind of traffic on the whole network at any time and conflicts with Policy1.

Even while specifying parameters for the network, users can choose conflicting values, as for example:

```
define link NPS_DARPA<NPS,DARPA>;

define link DARPA_SPAWAR<DARPA,SPAWAR>;

define path NPS_SPAWAR{NPS_DARPA,DARPA_SPAWAR};

define link_param NPS_DARPA{BW:= 150 MBPS};

define link_param DARPA_SPAWAR{BW:= 150 MBPS};

define path_param NPS_SPAWAR{BW:= 450 MBPS};
```

When the links NPS_DARPA and DARPA_SPAWAR are defined with a bandwidth of 150 Mbps each, there is no way path NPS_SPAWAR constructed from these links can be defined with a 500 Mbps connection, so a conflict exists. We also cannot create a policy with a *message* element if the path associated with that policy does not support the required *message* element.

The *user* attribute in the *target* item can also cause conflicts. Consider:

```
Policy6 net_manager {*} {traffic_class == {voice}{user == guven} {deny}

Policy7 net_manager {*} {traffic_class == {voice}{day != saturday } {permit}
```

Policy6 denies voice traffic from user 'guven' at any time. But the Policy7 permits voice traffic any day except Saturdays. Because no users are specifically mentioned, Policy7 covers all users and conflicts with Policy6.

Like users, restrictions may be applied to specific IP addresses by a PPL policy, which is another reason for a conflict:

```
Policy8 net_manager {*} {traffic_class == {voice}{131.120.10.*}{deny}

Policy9 net_manager {*} {*}{131.120.*.*}{permit}
```

Even though Policy9 permits any traffic from the IP address group 131.120.*.*, Policy8 denies voice traffic from a subnetwork of that IP address group.

3. Resolving Conflicts

As explained before, the first element of PPL policy format is the policy creator. In a PPL configuration file, users who can make policies must be defined. While listing user_id's, a priority level must be defined for each user. Here is an example:

```
define policy_maker Net_Manager(1), Xie(3), Stone(3), Guven(4);
```

When two conflicting policies have different creators, the priority levels of their creators are compared and the conflicts are resolved in favor of the higher-priority policy creator. When the policies have the same creator, or when the creators have the same priority level, automatic conflict resolution is not possible.

F. CHAPTER SUMMARY

This chapter includes general information about the PPL, the basic subject of this thesis. PPL has new features like path-based representation, support of both static and dynamic policies, support of future traffic classes, "message" support for existing and new network measurements, and the ability to scale well in large networks. A major contribution made by PPL is the ability to test for conflicting network policies and resolve them before they are disseminated throughout the network. We described the causes of PPL policy conflicts and how they are detected and resolved by the PPL compiler.

IV. PROBLEM DIAGNOSIS

A. THE PERFORMANCE OF THE PREVIOUS PPL COMPILER

Because [10] was focused on detecting and resolving conflicts correctly, the performance of the PPL compiler was not emphasized during the implementation of the compiler and policy tester [10].

Table 2 below shows the results of a test we made on five different simulated networks by using the compiler presented in [10]. The values in the table are total time to process and to test for conflicting policies. The time of PPL compiler was measured from the time of entering the PPL configuration file until printing out of conflict results. Sample networks differ by the number of nodes they have (10, 30, 40, 80 or 120) and the number of policies applied to them. For the test runs, a computer with a Pentium-IV 1.2 GHz. processor was used. The same computer will be used for testing the compiler improvement later in this thesis.

Number of Nodes	Num. of Policies	Num. of Paths	Compilation Time(seconds)
10	11	377	10
30	11	2271	120
40	20	8364	420
80	20	-	8
120	20	-	8

Table 2. Compilation test results with the previous PPL compiler.

Except for the 10-node network, all sample files use wildcard characters for representing paths, targets or conditions. The "8" sign next to 80 and 120-node policy files means that the compilation has not ended after 24 hours.

In [10], "scaling in large networks" was listed as one of the most important contributions of PPL. Thus these PPL compiler performance tests show a problem, that PPL is not feasible in networks having more nodes than 40. So our goal was to improve the performance of the PPL compiler and enable it to work for networks with hundreds of nodes.

B. THE CAUSES OF THE PERFORMANCE BOTTLENECK

In previous chapters it was mentioned that the conflict detection code of the PPL compiler consists mainly of two parts: parsing code written in C, and conflict detection code written in Prolog. Those two parts work hand and hand to compile the policies and detect the conflicts. The Prolog code is also divided into three parts. To find the problem, a step by step approach is performed for each code segment, starting with the parsing code in C.

Whatever the length of the sample file, the number of nodes in the network or the number of policies created, parsing noticeably ends in a few seconds. As a result the problem clearly was not in the C code. Therefore, the research was focused on the conflict detection code written in Prolog. For a better understanding, it is necessary to know what each Prolog stage does and which algorithms are being used at each stage.

1. Prolog Conflict Detection and Resolution Stages in Detail

As explained in preceding chapters, PPL can use the wildcard character in representing paths, which is easy for the user and a savings for policy storage. For example, the policy creators can just write "`*`" to represent all possible paths for the policies they want to apply to the whole network, or represent all paths between node A and node E just by writing `{A,*,E}`.

How this compression works is demonstrated in the sample partial network shown in Figure 6. The user wants to represent all *paths* from node A to node E. Rather than listing three separate paths, the user can write `{A,*,E}`. The advantage increases when there are additional paths between node A and node E.

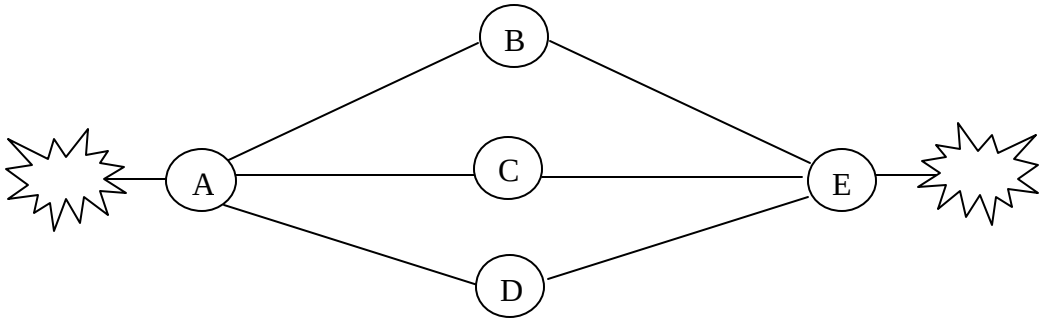

Figure 6. Representations by wildcard characters.

Other variations of using a wildcard character are possible. For example users who want to represent all paths using node C can write {*,C,*}. This "compression" definitely helps users while creating policies; however, the question remains as to which paths and nodes are compressed under those wildcard characters.

The original implementation has a two step approach for expanding policy paths with wildcard characters. First the compiler finds all possible paths in the network, and then selects the paths that match the policy path description. Stage 1 finds all possible paths in the network, prints the associations between policy labels, policy targets and policy conditions, and feeds these facts into the file "ppl2", which is the knowledge base for Prolog stage 2. To find all paths in the network between node A and node B, the network is searched by depth-first search first from A to B to find an acyclic path. All possible alternatives are tried along the path until all different paths have been found. (For example, Figure 7 shows two possible paths between two nodes labeled A and B.) To find all paths in the network, this algorithm must be repeated for all possible node pairs listed in the PPL configuration file.

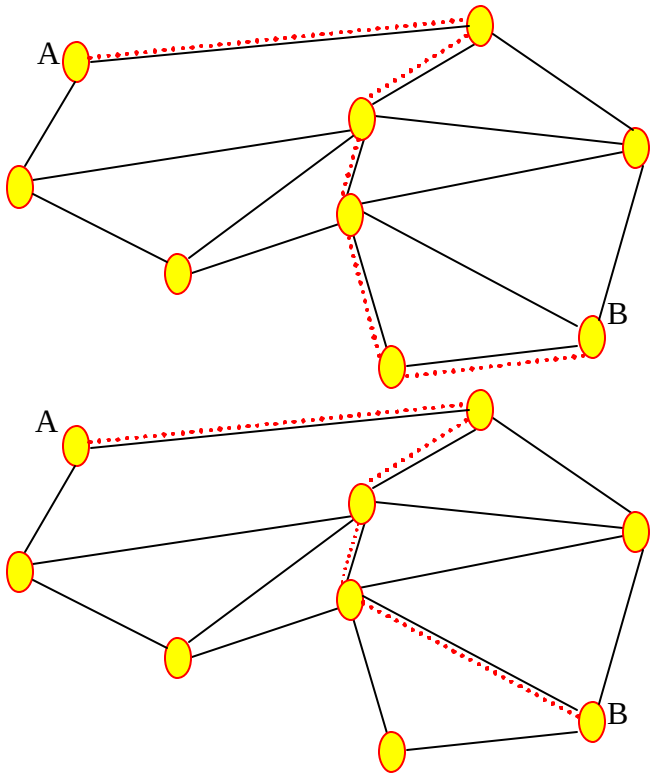

Figure 7. Two different paths detected between same nodes with the previous search algorithm.

Stage 2 examines the paths created by the first stage and eliminates those that are inconsistent with the given path specifications. For example, a policy with the path item {A,*,E} requires selection of paths starting with node A and ending with node E. Prolog stage 3 then detects conflicts of those overlapping policies. To find the overlapping paths, each path is compared against one another to find an overlapping path segment. That overlap could be either a node or link on the paths [10]. If an overlap is found, the target and conditional elements of each policy are compared. If they have compatible conditional elements and target traffic classes, then they conflict if their action items conflict. Some conflicts can be resolved by the PPL compiler described before.

2. The Performance Bottleneck

To analyze the performance bottleneck, each Prolog stage was run separately. Stage 1, when testing the 80 and 120-node sample file, ran for days without giving any result. Because a problem occurred as the number of nodes increased, it appeared that

performance is related to the number of paths required to be generated for wildcard matching. The flexibility of the path representation provided by PPL was a bottleneck because the path finding was not polynomial in time. But the wildcard symbol is very beneficial because it allows more flexibility in path representation, smaller policy databases, and less bandwidth to transmit path information to network enforcement points.

The Prolog code was found and examined. It can be seen in Appendix A that depth-first search is being used to find all paths between two nodes in the network. Considering that the complexity of the depth-first search for all paths between two nodes is $O(d^h)$ (where d is average number of children per state and h is the average depth), and also considering that this algorithm is being used by each pair of nodes, we get the complexity of $\binom{n}{2} x O(d^h)$ which is not polynomial. This suggests we find a better algorithm finding paths between two nodes.

THIS PAGE INTENTIONALLY LEFT BLANK

V. IMPROVING THE PPL COMPILER

A. RESOLVING THE PERFORMANCE BOTTLENECK

Finding all possible paths in a network is similar to the famous NP-complete Hamiltonian Cycle problem. This suggests it is also an NP-complete problem because there are 2^n possible paths in a fully connected network with n nodes. NP-complete problems have time complexity that is an exponential function of the problem size.

The first question is "Is it really required to find all possible paths in a network to find whether any two overlap?" Not necessarily; it's just simple to implement. Changing the code to find paths only between the nodes mentioned in policy paths is a valid restriction and not hard to implement. But that is not enough to solve our bottleneck problem because it does not reduce the exponential complexity of the algorithm.

1. A Bidirectional Search Algorithm

One idea is to use bidirectional search instead of depth-first search to find paths. The previous path finding started from a node trying to find a way to the other. However if the search works from both nodes at the same time, the depth of each search would be cut in half, so the exponent of the exponential of the search time would be cut in half, a dramatic saving. Because the new search algorithm is a better managed Prolog algorithm, the results for the small networks were very pleasing; the compilation of the same 40-node PPL configuration file took 4 minutes, almost two times faster than the previous path finding algorithm. But experiments with this idea on the 120-node network were not so satisfying.

We then modified the search to obey a depth limit. The algorithm travels forward and backward a number of steps up to an integer value the user assigns as *depth*. The number of returned possible paths and the time spent is related to the *depth* value the user chooses. If the user chooses a bigger *depth*, more paths will be returned, but it is no longer possible to get all solutions unless we pick an extremely large depth value.

Some experiments were done to find the optimum depth value to obtain an answer in a reasonable amount of time with correct conflict detection results. The new algorithm was also tested with different processor speeds to observe the effects on computation time. After these experiments, which are explained in Chapter VI, it was clear that finding all possible paths between two nodes in large networks with like 120 nodes took a lot of time. So we next focused on the question of whether we really must know all paths between two nodes in all cases to find conflicts with other paths.

2. Using Articulation Points

a. Using Articulating Points and Biconnected Components for Conflict Detection and Resolution

If a node breaks the graph into more than one disconnected component when removed it is called an *articulation point* [18]. This can be formalized by defining *a* as an articulation point of graph G if, and only if, vertices *v, w* exists; such that *v, w, a* are distinct and every path from *v* to *w* contains *a* [21]. If we remove all articulation points from a network, the network will be separated into its *biconnected components.* A graph G is called *biconnected* if for every distinct triple of vertices *v, w, a* there exists a path between *v* and *w* not containing *a* [22].

Even if it is impossible to know all paths represented by a wildcard character without expanding them, articulation points that divide a network into its biconnected components may help to make predictions without expansion. Consider the sample networks shown in Figure 8 and Figure 9. Figure 8 has three articulation points that are marked. Figure 9 is a network without articulation points except NATO. Assume the user defined paths path1={UN,*,NATO} and path2={NASA,*,IETF}. In Figure 8 the path {UN,*,NATO} represents only the link {UN,NATO}. Two different paths are represented by {NASA,*,IETF}: {NASA,IETF} and {NASA,NPS,DARPA,IETF}. As seen in the expanded path definitions, path1 and path2 have no physical overlap.

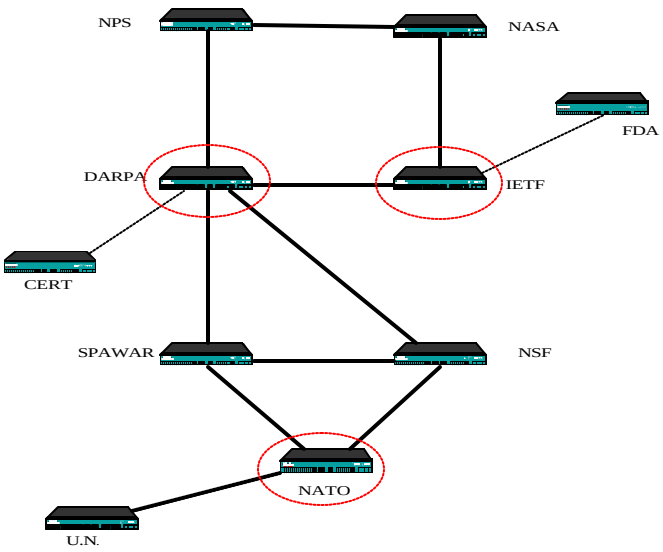

Figure 8. A sample network with marked articulation points.

Figure 9 is a modified form of Figure 8; a link is added between nodes IETF-NSF and nodes FDA and CERT are removed. In this form, IETF and DARPA are no longer articulation points. If we expand the same path definitions on this network, the result will be different. Even though {UN,*,NATO} represents the same link, there are more detected paths when we expand {NASA,*,IETF}. For example this time the paths {NASA,NPS,DARPA,SPAWAR,NATO,NSF,IETF}, {NASA,NPS,DARPA,NSF,IETF}, {NASA,NPS,DARPA,SPAWAR,NSF,IETF}, are also paths represented by {NASA,*,IETF}. This time path1 and path2 overlap.

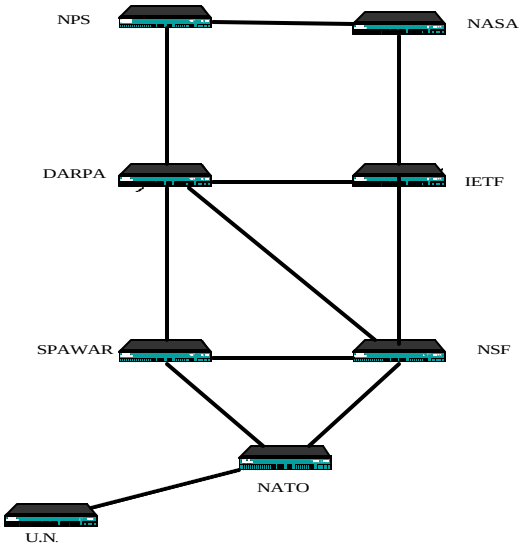

Figure 9. A sample network with only one articulation point.

When the number of articulation points decreases further and the network takes on a mesh topology, two policy paths which did not have overlapping links before turned out to be overlapping path definitions. This means that connectivity of the topology is an important issue in overlap detection. As a result, if the network can be separated into its biconnected components, it will be possible to make some guesses and simplifications while comparing paths. See the theorems in Appendix E.

3. A New PPL Conflict Detection and Resolution Process

We can apply the theorems in Appendix E to PPL conflict detection and resolution. Knowing the articulation points and biconnected components will let the compiler to make predictions when comparing two policy paths. The general case of two arbitrary policy paths is too complex to deal with here, but we can illustrate the necessary reasoning for one example. Assume there are two policy paths, `path1` and `path2`, that are defined by `{A,*,B}` and `{C,*,D}` respectively. There are five possibilities for nodes A, B, C and D.

1. A, B, C, and D are in the same biconnected component.
2. A and B are in a biconnected component and C and D are in a different component.

3. A and C are in a biconnected component and B and D are in a different component.
4. Three of the four nodes are in the same biconnected component, but not the fourth.
5. Each node is in a different biconnected component.

We analyze the above cases as follows.

Case 1. The first theorem in Appendix E proves that in a fully connected bidirectional topology, any link is a part of the representation {A,*,B} or {C,*,D}. If the directionality was not important, it is clear that there would be no need to expand the defined paths because of absolute overlap.

Nonetheless the link directions are important in PPL conflict detection and resolution process. For example, even though path {A,E,B} has a physical overlap with the link {E,B}, the link {B,E} is not considered as an overlap. The basic overlap elements are directed links, not nodes.

Because of the importance of direction of the links, in this case no prediction can be made about an overlap without expanding paths and comparing them for overlapping links.

There is an important issue to pay attention to when creating paths. All the path links between source and destination must belong to the biconnected component, because the biconnected component is disconnected from the rest of the network and both source and destination nodes are in the same biconnected component.

Such analysis enables using a limited number of nodes while searching for the possible paths. For this purpose, the depth limited bidirectional path search algorithm was revised. With this revised algorithm, to find the paths between two nodes, where both of the nodes are in the same biconnected component, node names and the members of the biconnected components are sent to the function at the same time. The algorithm selects only the nodes in the biconnected component list to create portions of paths within the component. Even though in some cases the depth limited bidirectional search algorithm must still be used, the number of nodes it must consider can be reduced. Since

the complexity of the NP-hard problem is directly related to the number of nodes in the network, the path finding process will be faster depending on the size of biconnected component.

Case 2. Due to the definition of *articulation point* and the *biconnected component,* all paths between any node pair in a biconnected component contain only nodes in the same biconnected component. This means paths {A,*,B} and {C,*,D} do not have an overlapping link. As a result, there is no need to expand and compare paths for overlap detection. There would be no overlap.

Case 3. In this case, both source nodes are in the same biconnected component and both destination nodes are together in another biconnected component. The paths going from source to destination must use articulation points as "gates" to visit another biconnected component. Since both paths searches are in the same direction (towards the articulation point to visit the biconnected component where source nodes are) for Theorem 1 of Appendix E, overlapping links will occur between two policy paths. Therefore in this case there is no need to expand paths. There is an overlap.

Case 4. This case is similar to Case 1. Due to the importance of the link directions, that no easy way exists to detect path overlaps. The compiler still has the advantage of using revised version of search algorithm which limits the number of nodes used in path expansion because at least one source and one destination node are in the same biconnected component.

Case 5. This is another case in which no shortcuts can be used. The compiler has to look for expanded versions of policy paths. After the expanded paths are detected, the policy paths are compared for physical overlap.

The cases that are not covered with the previous special cases will be handeled by the bidirectional search algorithm.

These above cases are for two policy paths both using the wildcard character. Similar guesses are also applied to situations where one of the two paths does not use the wild card character. The simplest case is when comparing a node with a path with the wildcard. If the node is in the same biconnected component with either of the source or destination node of the wildcard represented path, there is an overlap due to the first theorem of Appendix E On the other hand, if the source and destination node of the

wildcard path are in the same biconnected component that does not contain the node of the other path, there is not an overlap case due to the second theorem of Appendix E.

If one of the policy paths is a link and the other is a wildcard represented policy path, it is worthwhile to check whether the two end nodes of the first policy path are in one biconnected component while the source of and destination nodes of the wildcard path are together in a different biconnected component. The is because the condition is the same as Case 2 above, in which case there is no overlap.

4. Changing the Order in Conflict Detection and Resolution Method

Another improvement that we did to the previous implementation was to postpone the check for path overlap to the end of the policy comparison algorithm since it requires the most time. We first compare policy definitions for condition overlaps by a polynomial algorithm. Policies which have overlapping conditions are checked for target overlaps, which can also be done with a polynomial algorithm. Only if the policy definitions have both overlapping targets and overlapping conditions are they compared for path overlap.

5. A New Policy Conflict Detection Algorithm

Results of all these improvements were applied to the conflict-detection process by a new algorithm. The three step Prolog code approach of the previous compiler is kept for managing and debugging advantages.

The first stage starts by finding the biconnected components of the network topology under consideration. Then, the associations between policy labels, policy targets, and policy conditions are listed. All this information along with other information that will be used in the following steps (like node and link information) is written in a file to be used in Prolog stage 2.

The most important issue in Prolog stage 2 is making the associations between policy paths, policy conditions, and policy targets. After each policy label and the path definition are associated, the conditions and the targets that are related to that policy and

the path where it is applied are written to a file to use in Prolog stage 3. In addition, information about biconnected components and network links are forwarded to stage 3.

In Prolog stage 3, conflicts are detected and resolved as described at the end of the last section. If there is a conflict, the result is written to the "scan.out" file to let the policy creator know about the conflicts. Resolved conflicts are also printed to inform the users.

The new algorithm is as follows:

find_conflicts( )

```
1.  biconnected_components()  / find network's biconnected components
    for each (Policy1,Policy2) pair  / (Policy2,Policy1) will be checked later
2.      check_conditional_overlaps(Policy1,Policy2)  / compare it with
        others for conditional overlaps between policies
3.      if check_conditional_overlaps(Policy1,Policy2) == True
4.          check_target_overlaps(Policy1, Policy2)  / check the target
            overlaps2 between policies.
5.          if check_target_overlaps(Policy1, Policy2) == True
6.              check_physical_overlaps(Policyh1,Policy2)  / check
                physical overlaps between policies
7.              if check_physical_overlaps(Policyh1,Policy2) == True
8.                  print_conflict(Policy1,Policy2)
9.  return
```

The physical path overlap checking function takes advantage of the biconnected components as follows:

boolean check_physical _overlaps(Policy1,Policy2)

1. If Policy1→Path == "*" or Policy2→Path == "*"
2. return true / *"All paths" overlaps with any other path*
3. If Policy1→Path == Node1 Policy2→Path == Node2
4. return check_physical _overlaps1(Node1,Node2) / *Node vs. node*
5. If Policy1→Path == Node1 Policy2→Path == {A,*,B}
6. return check_physical _overlaps2(Node1,{A,*,B}) / *Node vs. wildcard path*
7. If Policy1→Path == Node1 Policy2→Path == List
8. return check_physical _overlaps3(Node1, List) / *Node vs. list of nodes*
9. If Policy1→Path == {A,*,B} Policy2→Path == {C,*,D}
10. return check_physical _overlaps4({A,*,B}, {C,*,D}) / *Wildcard vs. wildcard*
11. If Policy1→Path == {A,B} Policy2→Path == {C,*,D}
12. return check_physical _overlaps5({A,B}, {C,*,D}) / *Link vs. wildcard path*

The subroutines are as follows:

boolean check_physical _overlaps1(Node1,Node2)

1. if Node1 == Node2
2. return true

3. else

4. return false

boolean check_physical _overlaps2(Node1,{A,*,B})

1. if Node1 == A or Node1 == B

2. for i←1 to all Biconnected Components

3. if member ((Node1,A), Biconnected_Comonent(i)) ||
member ((Node1,B), Biconnected_Comonent(i)) / *Node1 is in the same biconnected component as either A or B.*

4. return true

5. if member((A,B), Biconnected_Component(i)), not member(Node1, Biconnected_Component(i)) / *A and B are in the same biconnected component that does not contain Node1.*

6. return false

7. return check_sublist(Node1,{A,*,B}) / *cannot be handled with previous conditions.*

boolean check_physical _overlaps3(Node1, List)

1. if member(Node1, List)

2. return true

3. else

4. return false

boolean check_physical _overlaps4({A,*,B}, {C,*,D})

1. if A == C, B == D
2. return true / *same paths*
3. for i←1 to all Biconnected Components
4. if member((A,C), Biconnected_Component(i)), B == D
5. return true / *Source nodes are in the same biconnected component and they have the same target node*.
6. if member((B,D), Biconnected_Component(i)), A == C
7. return true / *Source nodes are the same and destination nodes are in the same biconnected component*.
8. if member((A,C), Biconnected_Component(i))
9. for j←1 to All Biconnected Components, j != i
10. if member((B,D), Biconnected_Component(j))
11. return true / *Source nodes are in the same biconnected component while destination nodes are in another one*.
12. if member((A,B), Biconnected_Component(i)), not member((C,D), Biconnected_Component(i))
13. return false / *Source and destination of one path are in the same biconnected component, but not true for the other path.*
14. if member((A,B,C,D), Biconnected_Component(i))
15. limited_search_paths(A,B, Biconnected_Component(i),List1)
16. limited_search_paths(C,D, Biconnected_Component(i),List2)

17. return check_physical_overlaps(List1,List2) / *if all nodes are in the same biconnected component, call limited search algorithm which uses only the node that are member of biconnected component to expand the paths. List1 and List2 are expanded versions of path definitions. Compare returned paths.*

18. return check_sublist({A,*,B},{C,*,D}) / *if cannot be handled with previous conditions.*

boolean check_physical _overlaps5({A,B}, {C,*,D})

1. If A == C, B == D
2. return true
3. for i←1 to all Biconnected Components
4. if member((C,D), Biconnected_Component(i)), not member((A,B), Biconnected_Component(i))
5. return false
6. if member((A,B,C,D), Biconnected_Component(i))
7. limited_search_paths(C,D, Biconnected_Component(i),List1)
8. return check_physical_overlaps({A,B},List1)
9. return check_sublist({A,B},{C,*,D}) / *if cannot be handled with previous conditions*.

boolean check_sublist(List1,List2)

1. if sublist(List1,List2)
2. return true / *If List1 is a sublist of List2, they have a*

overlap.

6. Removing a Bug

While comparing user-defined policy paths, another concern is certifying that such paths really exist in the particular network. So the new compiler also checks this. But in addition, the previous compiler compares policies for subpaths based on only nodes and links. This means that only policy paths that are defined as a link or node are actually compared with other policies but not the ones defined by policy creators as list of nodes or the ones using wildcard characters. As a result of this bug, not all policies were compared to each other; so many conflicts could not be detected. So while improving the performance of the PPL compiler, we noticed and removed a bug causing major mistakes. This is illustrated by comparing the compiler result files in Appendix F and Appendix G, both the result of the compilation of the same sample file with 40 nodes; the new compiler detected all conflicts (Appendix G) that could not be detected by the previous compiler (Appendix F).

THIS PAGE INTENTIONALLY LEFT BLANK

VI. TEST RESULTS

For all the improvements described in the last section, 52 different Prolog rules were written comprising 255 lines of code.

A. THE NEW DEPTH-LIMITED SEARCH ALGORITHM

It is necessary to choose a *depth* value which will end the compilation in a reasonable time while returning an acceptable number of created paths between two nodes. Some experiments were made to choose the optimum depth value. Experiments were repeated 1000 times for the depths of 5 and 10; 500 times for a depth of 15; and 100 times for the depths of 20 and 25. Because it takes a long time to get results as the depth increases, experiments were repeated fewer number of times for higher depth values.

For each run two random nodes were selected as path endpoints. The number of detected paths between those two nodes (the number of paths the new search algorithm can create for that depth) and how long it took to find those paths were recorded. Figure 10 shows the number of returned paths (maximum, minimum and average) as we changed the *depth* parameter. Depth 20 appears to be a "saturation point" because there is little increase in the number of paths when we increase the depth to 25 from 20.

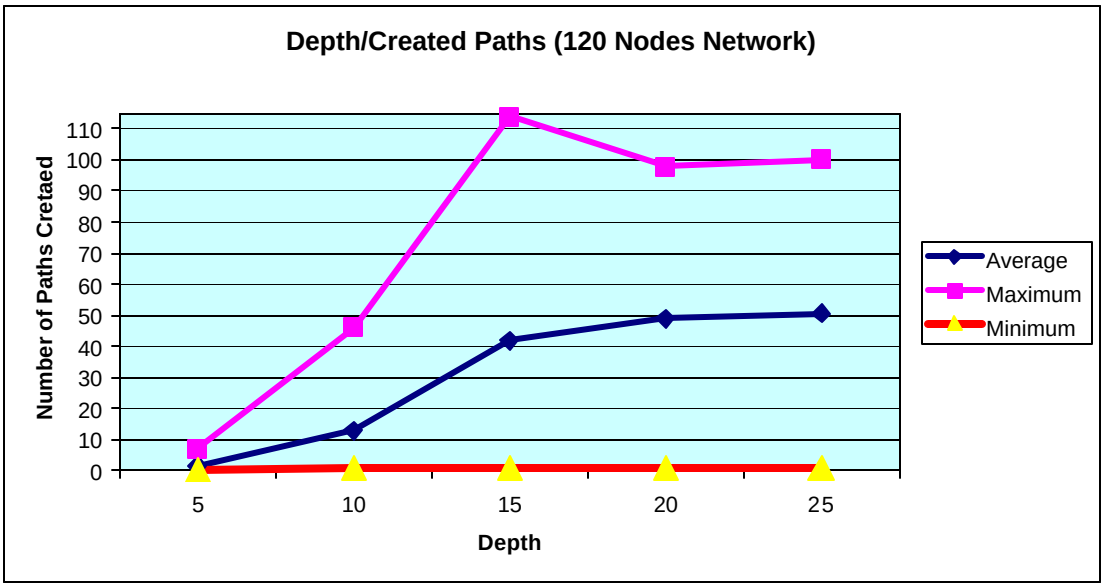

Figure 10. The effects of changing depth in the new path search algorithm to the number of paths detected.

Figure 11 below shows the time spent for finding paths between two random nodes as a function of depth value. The time in obtaining a result dramatically increases after depth 15. For actual data related to Figures 10 and 11, see Appendix B.

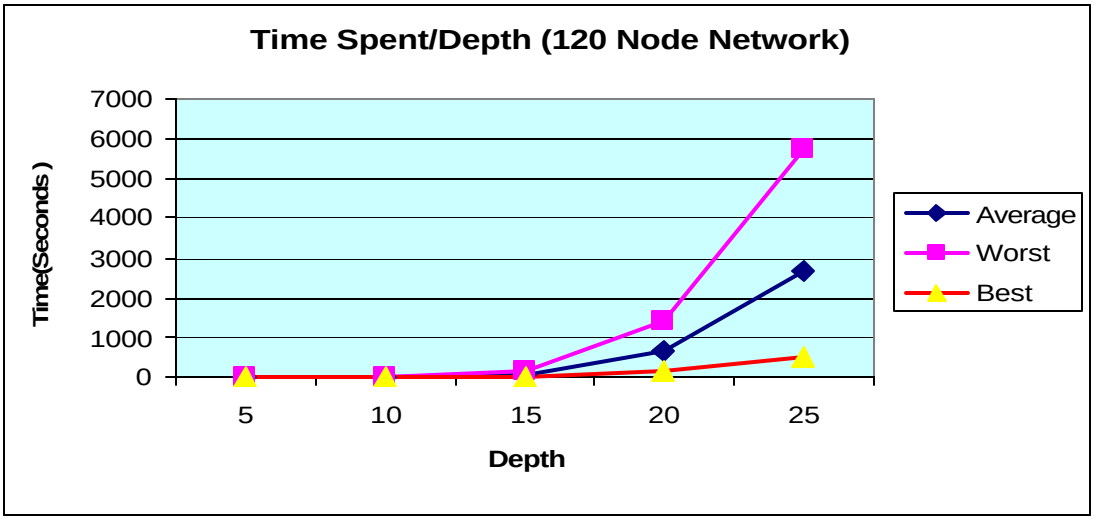

Figure 11. Time spent to return detected paths between two nodes.

Another issue is the effect of changing the processor speed. This time the code was run under the same conditions (random node selecting and same number of repeating), but on three different computers. The processors of the computers are Pentium-III 0.65 gigahertz, Pentium-IV 1.2 gigahertz and Pentium-IV 1.8 gigahertz. Figure 12 shows that the processor speed has no effect on the number of paths detected. The results on time spent (Figure 13) are encouraging because they show as the depth increased, the faster processors yield more benefits. The actual data related to Figure 12 and 13 can be found in Appendix C.

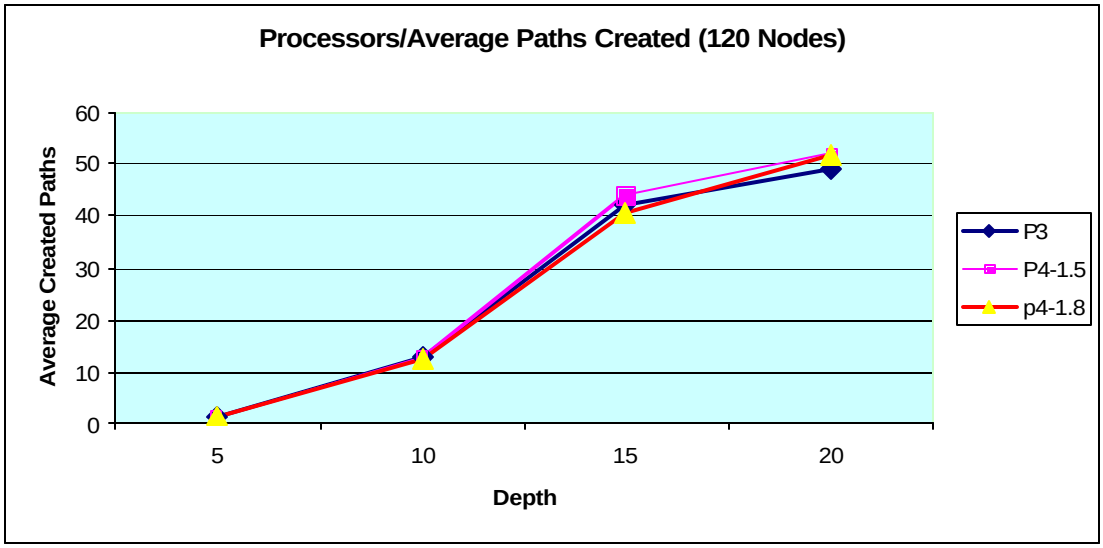

Figure 12. Number of returned paths by each processor between two random nodes.

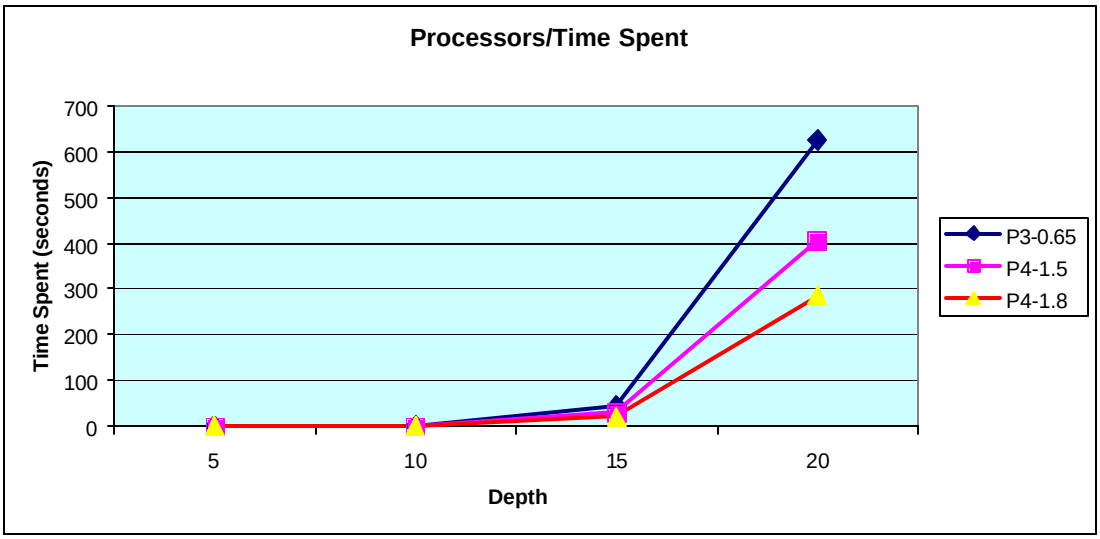

Figure 13. Time spent to find results by each processor.

B. COMPARING THE NEW COMPILER WITH THE OLD ONE

We compare the performance of the old and new PPL compilers in Table 3 and Figure 14. The improvement is impressive.

Number of Nodes	Num. of Policies	Compilation Time(seconds) using Previous Compiler	Compilation Time(seconds) using Improved Compiler
10	11	10	2
30	11	120	3
40	20	420	6
80	20	8	10
120	20	8	14

Table 3. Test results with previous and improved PPL compilers.

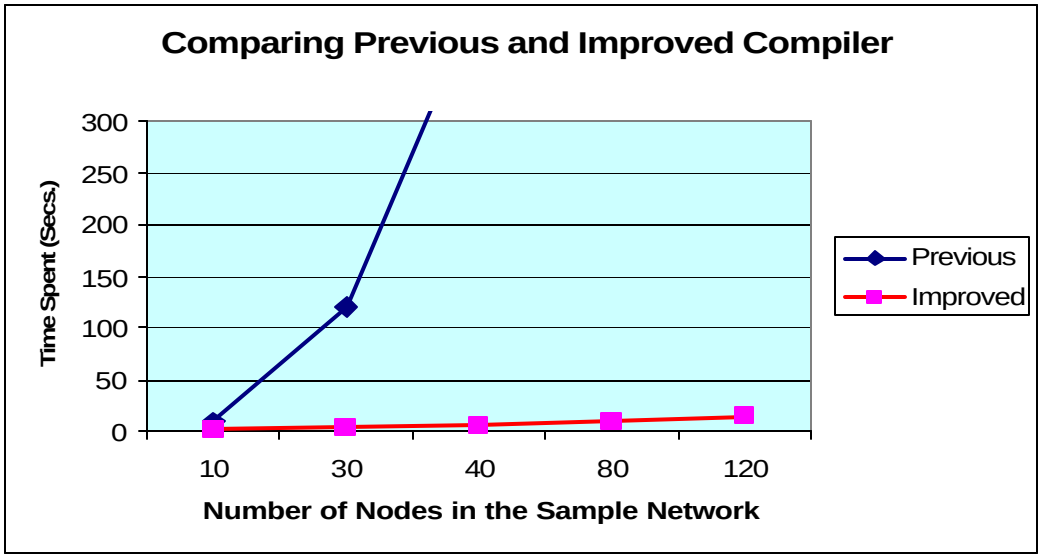

Figure 14. Comparing the previous and the improved PPL Compiler.

The results proved that the PPL compiler was enhanced enough to satisfy the time requirements to be used in real life networks. Moreover with the bug in the previous compiler removed the compiler was enabled to return all conflicts that actually exist between policies. Briefly, the new compiler returns more correct results in a much shorter time.

C. TEST RESULTS OF MORE DIFFICULT CASES

To make sure the new compiler is bug-free and works with any topology and with any number of wildcards in policy paths, new test files were created with more complex policy paths. First, several paths using the wildcard character were defined on a linear topology. Later, by adding random links, the topology was modified to a mesh. More paths with the wildcard character were added to policies. The test results can be seen in Table 4. For these experiments, policy paths were created by randomly choosing source and destination nodes. There were 11 defined polices in the 30-node sample file; the other files use 20 defined policies.

No. of Nodes in the Sample Network	No. of Policies	No. of Wildcard Paths	No. of Times Wildcard Paths Used in All Policies	Compilation Time (Seconds)	No of Times Physical Overlap Check Required
30 (Linear Topology)	11	8	17	3	1830
30 (Mesh Topology)	11	8	17	8	860
30 (Mesh Topology)	11	10	19	13	1570
40 (Linear Topology)	20	11	25	4	2795
40 (Mesh Topology)	20	11	25	8	2057
40 (Mesh Topology)	20	17	31	16	4488
80 (Linear Topology)	20	13	29	7	2726
80 (Mesh Topology)	20	13	29	105	3597
80 (Mesh Topology)	20	19	33	145	6354
120 (Linear Topology)	20	22	39	61	2422
120 (Mesh Topology)	20	22	39	2789	20210
120 (Mesh Topology)	20	26	42	3800	11861

Table 4. Testing more difficult cases

The first column in the table shows the number of nodes of the tested network. The second column is the number of wildcard symbols in policy paths. The third column shows the number of paths that uses the wildcard symbol "*" in their declarations. Such paths are termed wildcard paths. The third column shows the total number of times that wildcard paths are used in all policies. The value is higher because some policies are defined for several paths and a wildcard path may be used by multiple policies. For example consider the 80-node mesh topology which is presented in Appendix H. There are 13 wildcard paths as defined by `define path` statements. That value is shown in the second column. But many policies in that file are defined on several paths. For example `Policy1` is defined on paths `{n27_n25,n30_n70}` and `Policy2` is defined on paths `{n5_n28,n7_n37}`. The fourth column is the compilation time in seconds for that sample file. The fifth column shows the number of times the condition and target items overlapped between two policies, and hence the total number of times a physical overlap check was needed.

The table shows that as the number of wildcard path usage increased and as the number of the links in the network increased (from linear to mesh topology), the compilation time also increased. This increase is dramatic for large networks, like the 120-node network.

Another issue tested was the advantage of exploiting biconnected components in making the physical overlap check. Four situations occur with the improved PPL compiler:

1. The biconnected component property may permit a physical overlap decision to be made without a bidirectional path search.
2. The biconnected component property may permit a search on a reduced number of nodes belonging to only one biconnected component.
3. If the path overlap check has been done before for these policy paths and the result cached, we can recall that result.
4. If none of the above are possible, the full path search algorithm must be used on the entire network.

The results for each case are shown in the table below:

No. of Nodes in the Sample Network	No of Times Physical Overlap Check Required	No. of Times Biconnected Comp. Property Used	No. of Times Bidirectional Path Search Done with Reduced Network Size	No. of Times Cached Results Used	No. of Times Bidirectional Path Search Done on Entire Network
30 (Linear Topology)	1830	78	0	1746	8
30 (Mesh Topology)	860	102	1	750	7
30 (Mesh Topology)	1570	68	1	1493	9
40 (Linear Topology)	2795	126	0	2669	10
40 (Mesh Topology)	2057	249	1	1800	7
40 (Mesh Topology)	4488	145	7	4327	9
80 (Linear Topology)	2726	202	0	2511	13
80 (Mesh Topology)	3597	308	2	3276	11
80 (Mesh Topology)	6354	209	2	6127	16
120 (Linear Topology)	2422	151	0	2261	10
120 (Mesh Topology)	20210	478	1	19713	18
120 (Mesh Topology)	11861	332	2	11505	22

Table 5. Physical overlap check statistics.

Table 5 shows a dramatic decrease in using the bidirectional path search on the full network because of exploiting biconnected components. This is a big time saving especially in bigger networks. The results of the test can be summarized as follows:

- An important factor affecting the compilation time of the new compiler is the total number of times wildcard paths are used in all policies. This factor gives an upper bound on how many times an exponential-complexity search algorithm may have to be used. Exploiting biconnected components reduces the need of using the search algorithm but this may not help in some cases as when there is one large biconnected component and a few smaller ones.

- The nature of the network is an even more important factor. When the number of the links is increased, the time to perform a bidirectional search increases

rapidly due to its exponential complexity. Also, topology can matter considerably. A mesh-topology network usually has a large biconnected component instead of several similar sized ones, resulting in a much slower search than a less connected topology.

- We saw positive effects of exploiting biconnected components. In many cases, whether two policy paths overlap or not can be predicted without expanding their path representations.

VII. CONCLUSION AND FUTURE WORK

A. CONCLUSION

Before this thesis the PPL compiler could compile policies for networks with up to 40 nodes. The compilation for larger networks did not end even after days. This work identified and mitigated to a large extent the performance bottleneck of the previous PPL compiler.

The new compiler is very efficient and can return results for networks with 120 nodes and dozens of policy paths using the wildcard character. Moreover the improved compiler can detect all conflicts, which the previous compiler could not do because of a software bug.

The speed of the new compiler depends on the number of policy paths using the wildcard character and the number of nodes in the network. Each time a path with the wildcard character is tested against another path for overlap, the compiler may have to perform a bidirectional search of all paths between two nodes. The more nodes in the network, the depth of the search has to be larger to find all paths.

The topology of the network affects the complexity of the path overlap determination algorithm in another way. When the network can be partitioned into several biconnected components the algorithm is likely to succeed with no bidirectional search of all paths between two nodes. A mesh network with rich connectivity means few biconnected components, making it less likely to avoid a brute-force path search when determining if a path with the wildcard character overlaps with another path.

The test results show that with the improved compiler, PPL can be used for any topology of hundreds of nodes, with a reasonable number of policies and a reasonable number of policy paths. The advantages of the PPL system, as listed in Chapter III, can be more fully materialized after the improvements made by this work.

B. FUTURE WORK

Because this thesis initially focused on speeding up the PPL compiler, the solutions were only developed for the main path representation cases such as {A,*,E}. Users may want to declare more complex or more specific paths than the ones used as samples in this thesis. For example users may define a complex path declaration like {*,A,B,*,C,*}. It may also be desirable to support a negation operator (~) in path specification so that a path declaration ~{*,A,B,*} would represent all paths that do not go through nodes A and B in order. The improved PPL compiler cannot handle those path declarations. One area of future work may be enabling the PPL compiler to deal with these kinds of complex path representations. It must be kept in mind that biconnected components may be used to accelerate path overlap determination even with complex policy path representations. Methods minimizing the use of bidirectional search of all paths between two nodes in these cases must be developed.

Other methods may exist to reduce the use of bidirectional path search or limit the search space when a search is necessary. For example, it may be possible to utilize the information from the <conditions> element of a policy rule (such as a hop count constraint) to narrow the bidirectional path search space. Thus, another area of future work will be searching more ways to reduce the usage of the bidirectional search algorithm or narrow the search space. Such work should increase the PPL compiler performance further.

APPENDIX A. IMPORTANT ALGORITHMS FROM THE PREVIOUS PPL COMPILER

STAGE 1

1. list class{}, condition_path{}, link{}, no_conditions{}, node_label{}, path_links{}, path_message{}, path_param{}, policy_action{}, policy_message{}, policy_owner{}, policy_path{}, policy_targets{}, target{}, type{}, user{} / lists that are in Prolog database
2. list Policy_Paths{}, Target_lists{} / user defined lists
3. string user_implicit_deny, wild
4. open PPL2.txt / open the file to place the modified Prolog facts.
5. paths_in_nodes(&Policy_Paths{})
6. print_policy_list(Policy_Paths{})

   (5–6: output all paths and all the possible nodes that can be used to create the path)
7. create_paths()
8. policy_target_list(&Target_lists{})
9. print_target_list (Target_Lists{})

   (7–9: create an association between a policy and targets it supports.)
10. print_condition_list() / create a list of conditions and associate them with policies
11. explicit_nodes()
12. print_no_conditions()
13. print_implicit_deny()
14. print_path_params()
15. print_users()
16. print_actions()
17. print_types()
18. print_target_all()
19. print_policy_owner()
20. print_nodes()
21. print_links()
22. print_path_messages()
23. print_policy_messages()
24. close PPL2.txt / close the file

    (11–24: forward the important Prolog facts for Stage2)

//*
Given a Policy_Label a list will be returned with the first element being the Label, and the second element being the path associated with that label. ex: ['Policy1',['NPS','IETF']]
*//

paths_in_nodes(&Policy_Paths{})

1. List Expanded_Path{}
2. for i ← 1 to Length(path_links{})
3. { Path, Path_list{} } ← Nth (i, path_links{})
4. for j ← to Length (policy_path{})
5. { Policy_Label, Path1{} } ← Nth (j, policy_path{})
6. if Path1 = Path
7. path_in_nodes (Path_List{}, &Expanded_Path{})
8. Append (Expanded_Path{}, Policy_Paths{})

```
9.  for k ← 1 to Length( link{} )
10.     { Path, var1, var2 } ← Nth ( k, link{} )
11.     for l ←  to Length ( policy_path{} )
12.         { Policy_Label, Path1{} } ← Nth ( l, policy_path{} )
13.         if Path1 = Path
14.             path_in_nodes ( { Path }, &Expanded_Path{} )
15.             Append ( Expanded_Path{}, Policy_Paths{} )
16. for m ← 1 to Length( node_label{} )
17.     { Path, var } ← Nth ( i, node_label{} )
18.     for m ←  to Length ( policy_path{} )
19.         { Policy_Label, Path1{} } ← Nth ( m, policy_path{} )
20.         if Path1 = Path
21.             path_in_nodes ( { Path }, &Expanded_Path{} )
22.             Append ( Expanded_Path{}, Policy_Paths{} )
23. return
```

```
//*
```

Takes a path with link elements, and returns a path with those links expanded into its node components: wildcard characters are left untouched. Given: ['NPS_DARPA','DARPA_SPAWAR'] Returns: ['NPS', 'DARPA', 'SPAWAR']

```
*//

path_in_nodes( list1{}, &Expanded_path{} )

1.  for i ← 1 to Length( link{} )
2.      if  Head( list1{} ) = Head( Nth( i, link{} ) )
3.          Expanded_Path{} ← Tail( Nth( i, link{} )
4.   for j ← 1 to Length( node_label{} )
5.      if  Head( list1{} ) = Head( Nth(  j, node_label{} ) )
6.          Expanded_Path{} ← Tail( Nth(  j, node_label{} )
7.  if  Head( list1{} ) = ' * '
8.      if   Lengt( list1{} ) = 1
9.          Expanded_Path{} ← Head( list1{} )
10. for k ← 1 to Length( node_label{} )
11.     if  Head( list1{} ) = Head( Nth( k, node_label{} ) )
12.         list tail{} ← Tail( list1{} )
13.         list path{}
14.         path_in_nodes( tail{}, &path{} )
15.         Head( Expanded_Path{} ) ← Head( list1{} )
16.         Tail( Expanded_Path{} ) ← path{}
17.  if  Head( list1{} ) = ' * '
18.     list tail{} ← Tail( list1{} )
19.     list path{}
20.     path_in_nodes( tail{}, &path{} )
21.     Head( Expanded_Path{} ) ← Head( list1{} )
22.     Tail( Expanded_Path{} ) ← path{}
23.  return
```

//*

Generates the paths required to detect policy conflicts. If no wild card characters were used: then print just the paths explicitly listed and all the links that create the network else print out all possible paths through the network so that wild card matching can be done.

*//

create_paths()

1. list Nodes{}
2. if wild = ‘ no ‘
3. explicit_paths{}
4. explicit_links{}
5. if wild = ‘ yes ‘
6. for i ← 1 to Length(node_label{})
7. push (Nth (i, node_label{}), Nodes{})
8. create_paths(Nodes{})
9. return

//*

Return all possible paths incrementally by taking each node pair in the network and generating all possible paths between those to nodes.

*//

create_paths(Nodes{})

1. while Length(Nodes{}) > 0
2. node1 ← Head(Nodes{})
3. node2 ← Nth (2, Nodes{})
4. list tail{} ← Rest (2, Nodes{})
5. create_paths_helper ({ node1, node2, tail{} })
6. create_paths(node2, tail{})
7. return

//*

Find all possible paths between two nodes in the network

*//

create_pats_helper({ node1, node2, tail{} })

1. all_paths(node1, node2)
2. all_paths(node2, node1)
3. node2 ← Head(tail{})
4. tail{} ← Tail(tail{})
5. create_paths_helper({ node1, node2, tail{} })
6. return

```
//*
 Helper function to "all_paths". Determines if two nodes in the network are directly
connected
*//

path1( node1, node2{}, &Paths{} )

1.  list path{}
2.  path{} ← node2{}
3.  if  node1 = Head( path{} )
4.     Paths{} ← path{}
5.  else while adjacent( X, Head( path{} ) )
6.     if   member( X, path{} ) = false
7.        push( X, path{} )
8.        path1( node1, path{}, &Paths{} )
9.  return
```

```
//*
True if there is an link from X->Y or Y->X
*//

adjacent( X, y )

1.  boolean helper = true
2.  integer i = 1
3.  while helper = true and i =< Length ( link{} )
4.     { nodeX, nodeY } ← Nth( i, link{} )
5.     if nodeY = y
6.        if  nodeX is not assigned before
7.           X ← nodeX
8.           helper = false
9.     i = i + 1
10.  return X
```

```
//*
Remove duplicates from a list
*//
remove_dups( list1{}, &List2{} )

1.  head ← Head( list1{} )
2.  list tail ← Tail ( list1{} )
3.  while Legnth( list1{} ) > 0
4.     if    Member( head, tail{} ) = true
5.        remove_dups( tail{}, &List2{} )
6.     else
7.        push( head, List2{} )
8.        remove_dups( tail{}, &List2{} )
9.  return
```

```
//*
Create an association between a policy and the targets it supports.
*//

policy_target_list( &Target_List{} )
```

1. list targets{}, target_list{}, unique_target_list{}, results{}
2. for i ← 1 to Length(policy_targets{})
3. Label ← Head(i, policy_targets{})
4. targets{} ← Tail(i, policy_targets{})
5. create_target_list(Label, targets{}, &target_list{})
6. remove_dups(target_list{}, &unique_target_list{})
7. flatten(unique_target_list{}, &results{})
8. Tail(Target_list{}) ← results{}
9. Head(Target_list) ← Label
10. return

STAGE 2

1. link{}, no_conditions{}, node_label{}, path{}, path_message{}, path_param{}, policy_action{}, policy_condition{}, policy_message{}, policy_owner{}, policy_path{}, policy_targets{}, target{}, type{}, user{} / lists that are in Prolog database
2. string user_implicit_deny, wild
3. open PPL3.txt / open the file to place the modified Prolog facts.
4. all_policy_paths() — From all possible paths in the network, print only those associated with policies after the expansion of all wildcard characters.
5. print_no_conditions()
6. print_implicit_deny()
7. print_path_params()
8. print_users()
9. print_actions()
10. print_types()
11. print_target_all()
12. print_policy_owner()
13. print_nodes()
14. print_links()
15. print_path_messages()
16. print_policy_messages()
17. close PPL3.txt / close the file

(5–16: forward the important Prolog facts for Stage3)

```
//*
Print all paths associated with policies
*//

all_policy_paths()
```

```
1.  list Policy_Paths{}, Policy_Path{}
2.  for i ← 1 to Length( policy_path{} )
3.      { pol_label, pol_path{} ) ← Nth( i, policy_path{} )
4.      for j ← 1 to Length( path{} )
5.          pos_path{} ← Nth( j, path{} )
6.          policy_paths( pol_label, pol_path{}, pos_path{}, &Policy_Path{} )
7.          Append( Policy_Path{}, Policy_Paths{} )
8.  print_path_list( Policy_Paths{} )
9.  print_conditions( Policy_Paths{} )
10. return
```

```
//*
Match wild card paths, with their expanded paths
*//

policy_paths( pol_label, pol_path{}, pos_path{}, &Policy_Path{} )

1.  if   match( pol_path{}, pos_path{} )
2.      Head( Policy_Path{} ) ← pol_label
3.      Tail( Policy_Path{} ) ← pos_path{}
4.  return
```

```
//*
Does a given list with or without wildcard characters match another one?
*//

Boolean match( pol_path{}, pos_path{} )

1.  boolean returnValue
2.  if   Head( pol_path{} ) = Head( pos_path{} )
3.      match( Tail( pol_path{} ), Tail( pos_path{} ) )
4.  if   Head( pol_path{} ) = ‘ * ‘ and Head( pos_path{} ) = ‘ * ‘
5.      match( Rest( 2, pol_path{} ),  pos_path{} )
6.  if   Head( pol_path{} ) = ‘ * ‘ and Nth( 2, pol_path{} ) =  Head( pos_path{} )
7.      match( Tail( pol_path{} ), Tail( pos_path{} ) )
8.  if   Head( pol_path{} ) = Head( pos_path{} )
9.      match( Tail( pol_path{} ), Tail( pos_path{} ) )
10. if   pol_path{} = { Ø } and  pos_path{} = { Ø }
11.     returnValue = true
12. if   Length( pol_path{} ) = 0 and  pos_path{} = { Ø }
13.     returnValue = false
14. if   pol_path{}  = ‘ * ‘
15.     returnValue = true
16. return returnValue
```

STAGE 3

1. condition{}, link{}, no_conditions{}, node_label{}, path{}, path_message{}, path_param{}, policy_action{}, policy_message{}, policy_owner{}, target{}, type{}, user{} / lists that are in Prolog database
2. list subpaths{}, permit_conflicts{} / user defined lists
3. string user_implicit_deny
4. open scan_out.txt / open the file to place the modified Prolog facts.
5. find_all_subpaths(&subpaths{})
6. permit_conflicts(subpaths{}, &permit_conflicts{})
7. print_unresolved_conflict_list(permit_conflicts{})
8. print_resolved_conflict_list(permit_conflicts{})
9. print_message_conflict_list()
10. close scan_out.txt / close the file

```
//*
Return set of all policies that contain overlapping paths.
*//

find_subpaths( &subpaths{} )

1.  for i ← 1 to Length( path{} )
2.      { policy1, path1{}, target1 } ← Nth( i, path{} )
3.      for j ← 1 to Length( link{} )
4.          { var, from, to } ← Nth( j, link{} )
5.          if   { from, to } = path1{}
6.              for k ← 1 to Length( path{} )
7.                  { policy2, path2{}, target2 } ← Nth( k, path{} )
8.                  if    sublist( path1{}, path2{} ) = true
9.                      Push( { policy1, path1{}, target1, policy2, path2{},
                                    target2 },  subpaths{} )
10. for l ← 1 to Length( path{} )
11.     { policy1, path1{}, target1 } ← Nth( l, path{} )
12.     for m ← 1 to Length( node_label{} )
13.         { label, var } ← Nth( m, node_label{} )
14.         if   {label} = path1{}
15.             for n ← 1 to Length( path{} )
16.                 { policy2, path2{}, target2 } ← Nth( n, path{} )
17.                 if    sublist( path1{}, path2{} ) = true
18.                     Push( { policy1, path1{}, target1, policy2, path2{},
                                    target2},  subpaths{} )
19. return
```

```
//*
Checks whether  list1{} a sublist of  list2{}
*//

sublist( list1{}, list2{} )

1.  while Length( list2{} ) > 0
2.      returnValue = true
3.      for i ← 1 to Length( list1{} )
4.          if   Nth( i, list1{} ) != Nth( i, list2{} )
5.              retutnValue = false
6.      sublist( list1{}, Tail( list2{} ) )
7.  return returnValue
```

```
//*
```

Take a list of policy pairs that contain overlapping paths. Check each policy pair for overlapping conditions. If the conditions do overlap, then check the classes of traffic that are permitted on each to determine if a conflict exists.

```
*//

permit_conflicts( subpaths{}, &permit_conflicts{} )

1.  list no_overlap{}, conflicts{}
2.  for i ← 1 to Length( subpaths{} )
3.      { policy1, path1{}, target1{}, policy2, path2{}, target2{} } ← Nth( i,
            subpaths{} )
4.      list tail{} ← Tail( subpaths{} )
5.      if   policy1 != policy2
6.          conditional_overlap( policy1, policy2, var{}, &no_overlap{} )
7.          if   no_overlap{} != { Ø }
8.              conflict_permit_targets( policy1, policy2, target2{},  target1{},
                                                &conflicts{} )
9.              if   conflicts{} != { Ø }
10.                 Push( { policy1, path1{}, target1{}, policy2, path2{},
                                target2{}, conflicts{} }, permit_conflicts{} }
11.                 permit_conflicts( tail{}, &permit_conflicts{} )
12.             for j ← 1 to Length( target{} )
13.                 { policyJ, targetJ } ← Nth( j, target{} )
14.                 if   policyJ = policy1 and targetJ{} = ‘ permit_all ‘
15.                     for k ← 1 to Length( policy_action{} )
16.                         { policyK, actionK } ← Nth( k, policy_action {} )
17.                         if   policyK = policy2 and actionK = ‘ deny ‘
18.                             value ← ‘ permit_all ‘
19.                             Push( { policy1, path1{}, target1{}, policy2, path2{},
                                            target2{}, { policy1, value } }, permit_conflicts{} }
20.                         permit_conflicts( tail{}, &permit_conflicts{} )
21.                 if   policyJ = policy1 and targetJ{} = ‘ deny_all ‘
22.                     for l ← 1 to Length( policy_action{} )
23.                         { policyL, actionL } ← Nth( l, policy_action{} )
24.                         if   policL = policy2 and actionK = ‘ permit ‘
```

```
25.                         value ← ‘ deny_all ‘
26.                         Push( { policy1, path1{}, target1{}, policy2, path2{},
                                    target2{}, { policy1, value } }, permit_conflicts{} }
27.                        permit_conflicts( tail{}, &permit_conflicts{} )
28.             for m ← 1 to Length( target{} )
29.                 { policyM, targetM } ← Nth( m, target{} )
30.                 if   policyM = policy2 and targetM{} = ‘ permit_all ‘
31.                     for n ← 1 to Length( policy_action{} )
32.                         { policyN, actionN } ← Nth( n, policy_action {} )
33.                         if   policyN = policy1 and actionN = ‘ deny ‘
34.                             value ← ‘ permit_all ‘
35.                             Push( { policy1, path1{}, target1{}, policy2, path2{},
                                        target2{}, { policy2, value } }, permit_conflicts{} }
36.                            permit_conflicts( tail{}, &permit_conflicts{} )
37.                 if   policyJ = policy2 and targetJ{} = ‘ deny_all ‘
38.                     for o ← 1 to Length( policy_actio n{} )
39.                         { policyO, actionO } ← Nth( o, policy_action{} )
40.                         if   policO = policy1 and actionO = ‘ permit ‘
41.                             value ← ‘ deny_all ‘
42.                             Push( { policy1, path1{}, target1{}, policy2, path2{},
                                        target2{}, { policy2, value } }, permit_conflicts{} }
43.                            permit_conflicts( tail{}, &permit_conflicts{} )
44.             else if conflicts{} = { Ø }
45.                 permit_conflicts( tail{}, &permit_conflicts{} )
46.     if   policy1 = policy2
47.         permit_conflicts( tail{}, &permit_conflicts{} )
48.     if   subpaths{} = { Ø }
49.         permit_conflicts{} ← { Ø }
50. return
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. TEST RESULTS OF BIDIRECTIONAL SEARCH ALGORITHM WITH DIFFERENT DEPTH VALUES

(RESULTS FOR 120-NODE NETWORK)

NUMBER OF DETECTED PATHS			
AV.	MAX	MIN	DEPTH
1.559727	7	0	5
12.92776	46	1	10
41.86842	114	1	15
48.85714	98	1	20
50.7722	100	1	25

Table 6. Number of detected paths for each depth value

TIME FOR CREATING PATHS BETWEEN TWO NODES								
COMPUTATION(1/SEC)						TIME(SEC)		
AV.	MAX.	MIN.		DEPTH		AV	MAX.	MIN.
17.6597	4.545455	100		5		0.056626	0.22	0.01
0.677684	0.185874	4.545455		10		1.475614	5.38	0.22
0.022109	0.007185	0.114416		15		45.23125	139.18	0.22
0.001598	6.91E-04	0.006424		20		625.8659	1448.16	155.66
3.73E-04	1.73E-04	0.001933		25		2682.29	5772.29	517.264

Table 7. Time spent to detect paths for each depth value

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. TEST RESULTS OF BIDIRECTIONAL SEARCH ALGORITHM WITH DIFFERENT PROCESSOR SPEEDS

(RESULTS FOR 120-NODE NETWORK)

AVARAGE DETECTED PATHS				
P3-0.65	P4-1.5	P4-1.8		DEPTH
1.559727	1.559006	1.548117		5
12.92776	12.97143	12.75		10
41.86842	43.98621	40.35739		15
48.85714	51.9635	51.57463		20

Table 8. Average path detected for each processor with different depths.

TIME FOR PATH DETECTION BETWEEN TWO NODES								
COMPUTATION(1/SEC)						TIME(SEC)		
P3-0.65	P4-1.5	P4-1.8		DEPTH		P3	P4-1.5	P4-1.8
17.6597	27.06539	32.2318		5		0.056626	0.036948	0.031021
0.677684	1.072781	1.425857		10		1.475614	0.932157	0.701333
0.022109	0.031674	0.045678		15		45.23125	31.57174	21.89241
0.001598	0.002468	0.0035		20		625.8659	405.1912	285.7387

Table 9. Time spent to detect paths with each processor.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. PROLOG CONFLICT DETECTION AND RESOLUTION CODE OF THE IMPROVED PPL COMPILER

```
% Stage1 is the first step in determining policy conflicts.
% Steps:  * Open "ppl2.txt" file for policy conflict information
%         * Separate network to its biconnected components, and
%         label each biconnected component.
%         * Print the association between policy labels and
%           policy targets.
%         * Print the association between policy labels and
%           policy conditions.
%         * Copy all other necessary facts from stage one to
%           the file that will be used in stage two.
%         * Close the output file
%
% Authors: Neil ROWE, Gary STONE, Ahmet GUVEN
%
stage1:-
        %
        % Open the file ppl2.txt to place the modified Prolog facts
        % to be used in stage two.
        %
        open('ppl2.txt',write,output),
        set_output(output),

        % Divide network to its biconnected components by finding
        % articulation points.
        %
         artics,

        %
        % Output paths & all the possible nodes that can be used
        % to create the path.
        %
        setof(Policy_Path,paths_in_nodes(Policy_Path),Policy_Paths),
        not(print_policy_list(Policy_Paths)),

        %
        % Create an association between a policy and the targets
        % it supports.
        %
        setof(Target_List,policy_target_list(Target_List),Target_Lists),
        not(print_target_list(Target_Lists)),
```

```
        %
        % Create a list of conditions and associate them with policies
        %
        not(print_condition_list),

        %
        % Forward the facts of users, bandwidth, no_conditions, actions and types
        % to the file for stage two.
        %
        not(print_no_conditions),
        print_implicit_deny,
        not(print_path_params),
        not(print_users),
        not(print_actions),
        not(print_types),
        not(print_target_all),
        not(print_policy_owner),
        not(print_nodes),
        not(print_links),
        not(print_path_messages),
        not(print_policy_messages),

        %
        % Close the output file
        %
        set_output(user_output),
        close(output),
        write('Stage 1 complete'),nl,nl.


%=====================================================
% This rule helps finding articulation points in the network. Calls the main
% rule to find articulation points and biconnected components. After biconnected
% components found, it forwards the marked biconnected components as a list
% to the second stage.
%
:- dynamic articulation/1, nodenum/1, low/2.
:- style_check(-singleton).
:- unknown(_, fail).

artics :- retractall(node(_,_,_)), retractall(low(_,_)),
        retractall(bic_list(_,_)),
        retractall(articulation(_)),retractall(stack(_)),
        link(_,Start,_), !, retractall(nodenum(_)),
        assertz(nodenum(0)),retractall(counter(_)),assertz(counter(0)),
        assertz(stack([])), dfs(Start,0),artics(Start,[]),!,
```

```
        listing(bic_list),nl.


%=======================================================
% Helps to find articulation points and biconnected components.
% visits the nodes by Depth First Search and assigns each node
% a Depth First Search Number.
%
dfs(Node,Back) :- nodenum(K), retract(nodenum(K)), Kp1 is K+1,
        assertz(nodenum(Kp1)), assertz(node(Node,Back,Kp1)),
        xlink(Node,Node2), not node(Node2,_,_), dfs(Node2,Node), fail.

dfs(_,_).

%=======================================================
% This is the main rule to find articulation points and biconnected
% components. While finding articulation points, visited nodes are
% pushed in stack. When the articulation point is found, the nodes
% in the stack are labeled as its biconnected components.
%
artics(Node,Stack) :- node(Node2,Node,_),push1(Node),push1(Node2),
         artics(Node2,Stack1),fail.

artics(Node,Stack) :- node(Node,_,Num),
        nice_bagof(Num2,backward_xlink_num(Node,Num2),NL2),
        nice_bagof(Num3,child_low(Node,Num3),NL3),
        append([Num|NL2],NL3,NL), minlist(NL,Low),
        assertz(low(Node,Low)),ifthen((node(Node,Back,_),
        node(Back,_,Num1),low(Node,Low), Low >= Num1),
        print_all(Back)).

%=======================================================
% Helper function to find articulation points and biconnected
% components. Checks whether nodes are connected by a link.
%
xlink(Node1,Node2) :- link(_,Node1,Node2); link(_,Node2,Node1).


%=======================================================
% Helper function to find articulation points and biconnected
% components.
%
backward_xlink_num(Node,Num) :- xlink(Node,Node2),
        not node(Node,Node2,_), not node(Node2,Node,_),
        node(Node2,_,Num).
```

```
%=========================================================
% Helper function to find articulation points and biconnected
% components.
%
child_low(Node,Low) :- node(Node2,Node,_), low(Node2,Low).

%=========================================================
% Helper function to find articulation points and biconnected
% components.
%
nice_bagof(A,B,C) :- bagof(A,B,C), !.

nice_bagof(_,_,[]).

%=========================================================
% Helper function to find articulation points and biconnected
% components.
%
minlist([X],X) :- !.

minlist([X|L],M) :- minlist(L,M), M<X, !.

minlist([X|_],X).

%=========================================================
% This is the main rule to list the biconnected components.
% Calls a subroutine to deal with special cases of
% biconnected components and articulation points.
%
print_all(Back):- retractall(counter(_)),assertz(counter(0)),
       assertz(articulation(Back)),list_and_print(Back),!.


%=========================================================
% This rule first removes the duplicates in the stack that visited nodes
% are collected. Then calls another rule checking whether articulation point
% should be included in the biconnected components or not.
%
list_and_print(Back):- stack(New_Stack),
          reverse_and_remove_dups(New_Stack, Biconnected_Components),
          check_bic(Back,Biconnected_Components,New_Biconnected),
          print_stack(Back,New_Biconnected, []).
```

```
%=======================================================
% Removes nicely the duplicates in the stack where visited nodes are
% collected.
%
reverse_and_remove_dups(List1,List2):-reverse(List1,List3),
        remove_dups(List3,List4),reverse(List4,List2).

%=======================================================
% This rule decides that whether the articulation point must be included
% to biconnected components or not. If it should not, it romeves it from the
% stack.
%
check_bic(Back,Biconnected_Components,New_Biconnected):-
        nth1(1, Biconnected_Components, Node1),
        nth1(2, Biconnected_Components, Node2),stack(Biconnecteds),
        reverse_and_remove_dups(Biconnecteds, Biconnected),
        ifthenelse(Node2=@=Back,
        (ifthenelse((articulation(Node1),used(Node1)),
        (pop(Biconnected, Biconnec),pop1(Biconnec),
        stack(Bicon),New_Biconnected=Bicon),
        New_Biconnected= Biconnected)),New_Biconnected= Biconnected).

%=======================================================
% Helper function to find articulation points and biconnected
% components. After the decision made about the articulation point,
% the biconnected components are listed.
%
print_stack(Node,[Node|Rest],Storing_List):-counter(K),
        ifthenelse( (not K=@=1,not K=@=0),(In_List = [Node|Storing_List],
        assertz(bic_list(Node,In_List))),
        assertz(bic_list(Node,Storing_List))),
        retractall(counter(_)),assertz(counter(0)).

print_stack(Node,[Head|Rest],Storing_ List):-not Node=@=Head,
        counter(K), retract(counter(K)), Kp1 is K+1, assertz(counter(Kp1)),
      pop1(Rest),print_stack(Node,Rest,[Head|Storing_List]).

print_stack(Node,[Head|Rest],Storing_List).

%=======================================================
% This rule helps deciding whether the articulation point must be included
% to biconnected components or not.
%
used(Node):-bic_list(_,List), member(Node,List),!.
```

```
%=======================================================
% Helper function to find articulation points and biconnected
% components. Pops an element from the stack.
%
pop([Head_Stack|Tail_Stack], New_Stack):- New_Stack=Tail_Stack.

%=======================================================
% Helper function to find articulation points and biconnected
% components. Pushs an element to the stack.
%
push(Element,Stack, New_Stack):- New_Stack=[Element|Stack].

%=======================================================
% Helper function to find articulation points and biconnected
% components. Dynamic Pop. Deletes the old list and asserts the
% popped list.
%
pop1(Rest):- retract(stack(S)),assertz(stack(Rest)).

%=======================================================
% Helper function to find articulation points and biconnected
% components. Dynamic Push. Deletes the old list and asserts the
% pushed list.
%
push1(Element):- retract(stack(S)),New_Stack=[Element|S],
        assertz(stack(New_Stack)).

%=======================================================
% Helper function to find articulation points and biconnected
% components. Represents If-Then in Prolog
%
ifthen(P,Q):-call(P),!,call(Q).

ifthen(_,_).

%=======================================================
% Helper function to find articulation points and biconnected
% components. Represents If-Then-Else in Prolog
%
ifthenelse(P,Q,R):-call(P),!,call(Q).

ifthenelse(P,Q,R):-call(R).
```

```
%=========================================================
% Prints all paths explicitly listed by the user
%
explicit_paths:- setof(Path, path_links(_, Path), Paths),
        print_path_list(Paths),fail.

%=========================================================
% Prints all possible links in the network
%
explicit_links:- findall(Link, link(Link,_,_), Links),!,
        remove_dups(Links, Links_nodup),!,
        print_link_list(Links_nodup),!,fail.

%=========================================================
% Prints all the nodes in the network
%
explicit_nodes:- setof(Node, node_label(Node,_), Nodes),
        print_node_list(Nodes),fail.

%=========================================================
% Prints out the paths associated with each policy
%
paths:- setof(Path,paths_in_nodes(Path),Paths),
        qsort(Paths,Sorted),
      not(print_list(Sorted)).

%=========================================================
% Prints out a node list nicely formatted
%
write_path([]):- write(']').

write_path([X|[]]):- term_to_atom(X,X_atom),
        write(X_atom),
        write(']'),!,true.

write_path([X|Tail]):- term_to_atom(X,X_atom),
        write(X_atom),
        write(','),
        write_path(Tail).

%=========================================================
% Prints out a path represented as a list of nodes
%
print_node_list([]):-fail.

print_node_list([Link|Tail]):-
```

```
        node_label(Link,_),
        write('path(['),
        term_to_atom(Link,Link_atom),
        write(Link_atom),
        write(']).'),nl,
        print_node_list(Tail),fail.

%=======================================================
% Prints out all the links in the network
% nicely formatted.
%
print_link_list([]):- fail.

print_link_list([Link|Tail]):-
        link(Link,Src,Dst),!,
        term_to_atom(Src,Src_atom),
        term_to_atom(Dst,Dst_atom),
        write('path(['),
        write(Src_atom),
        write(','),
        write(Dst_atom),
        write(']).'),nl,
        write('path(['),
        write(Dst_atom),
        write(','),
        write(Src_atom),
        write(']).'),nl,
        print_link_list(Tail),!,fail.

%=======================================================
% print_path_list:
% Outputs a list of lists, where each
% list is on a line by itself and
% all the atoms are quoted.

print_path_list([]):- fail.

print_path_list([Path | Tail]) :-
         not(eq1(Path)),
         write('path(['),
         write_path(Path),
         write(').'), nl, print_path_list(Tail).

print_path_list([[X|Y] | Tail]) :-
         eq1([X|Y]),
         print_path_list(Tail).
```

```
print_path_list([[X|Y] | Tail]) :-
        eq1([X|Y]),
        node_label(X,_),
        write('path(['),
        term_to_atom(X,X_atom),
        write(X_atom),
        write(']).'),nl,
        print_path_list(Tail).

%=========================================================
% print_policy_list:
% Outputs nicely for each policy the paths
% associated with the that policy
%
print_policy_list([]):- fail.
print_policy_list([[Label|[Path]] | Tail]) :-
        write('policy_path('),
        term_to_atom(Label,Label_atom),
        write(Label_atom),
        write(',['),
        write_path(Path),
        write(').'), nl, print_policy_list(Tail).

%=========================================================
% Print out a list of targets associated with a policy
%
write_target_path([X|[]]):- term_to_atom(X,X_atom),
        write(X_atom),!,true.

write_target_path([X|Tail]):- term_to_atom(X,X_atom),
        write(X_atom),
        write(','),
        write_target_path(Tail).

%=========================================================
% Prints out the association between a policy
% and its targets
%
print_target_list([[Label|Targets] | Tail]) :-
        write('policy_target('),
        term_to_atom(Label,Label_atom),
        write(Label_atom),
        write(','),
        write_target_path(Targets),
        write(').'), nl,
```

```
        print_target_list(Tail).

%=======================================================
% Given a Policy_Label a list will be returned
% with the first element being the Label, and
% the second element being the path associated
% with that label
% ex: ['Policy1',['NPS','IETF']]
%
paths_in_nodes([Policy_Label|[Expanded_Path]]):-
        path_links(Path,Path_List),
        policy_path(Policy_Label,Path),
        path_in_nodes(Path_List,Expanded_Path).

paths_in_nodes([Policy_Label|[Expanded_Path]]):-
        link(Path,_,_),
        policy_path(Policy_Label,Path),
        path_in_nodes([Path],Expanded_Path).

paths_in_nodes([Policy_Label|[Expanded_Path]]):-
        node_label(Path,_),
        policy_path(Policy_Label,Path),
        path_in_nodes([Path],Expanded_Path).

%=======================================================
% Takes a path with link elements, and returns a path
% with those links expanded into its node
% components: wildcard characters are left
% untouched.
%
% Given: ['NPS_DARPA','DARPA_SPAWAR']
% Returns: ['NPS', 'DARPA', 'SPAWAR']
%
path_in_nodes([X],[N1,N2]):- link(X,N1,N2).

path_in_nodes([X],[X]):- node_label(X,_).

path_in_nodes([X],[X]):- X = '*'.

path_in_nodes([X|Tail],[X|Path]):-
        node_label(X,_),
        path_in_nodes(Tail,Path).

path_in_nodes([X|Tail],[X|Path]):-
        X = '*',
        path_in_nodes(Tail,Path).
```

```
%=========================================================
%Prints a list.
%
print_list([]):- fail.

print_list([X | Tail]) :- write(X), nl, print_list(Tail).

%=========================================================
% Uses quicksort to sort a list of lists by the
% number of elements in each list
qsort( [], []).

qsort([X | Tail], Sorted) :-
        split( X, Tail, Small, Big),
        qsort( Small, SortedSmall),
        qsort( Big, SortedBig),
        conc( SortedSmall, [X | SortedBig], Sorted).

%=========================================================
% Helper function to qsort, splits one list
% into two parts
%
split( _, [], [], []).

split( X, [Y | Tail], [Y | Small], Big) :-
        gt( X, Y),!,
        split(X, Tail, Small, Big).

split(X, [Y | Tail], Small, [Y | Big]) :-
        split(X, Tail, Small, Big).

%=========================================================
% Is X is greater than Y
%
gt(X, Y) :- length(X, Xlen) , length(Y, Ylen), Xlen > Ylen.

%=========================================================
% Returns the first element of a list
%
first([],[]).

first( X, [X | _ ]).
```

```
%=========================================================
% Given list of traffic classes, expand list
% by assigning action and operation, to
% each class in the list.
%
expand_value_list(_,Action,Class,Op,List,[Expanded_List]):-
        expand_list(Action,Class,Op,List,Expanded_List).

%=========================================================
% Helper to "expand_value_list"
% Assign action, and opertion to each value
% defined for the traffic class.
%
expand_list(_,_,_,[],[]).

expand_list(Action,Class,Op,[X|Tail],[Action,Class,Op,X|Results]):-
        expand_list(Action,Class,Op,Tail,Results).

%=========================================================
% Removes duplicates from a list
%
remove_dups([],[]).

remove_dups([Head|Tail],List):-
        member(Head,Tail),
        remove_dups(Tail,List).

remove_dups([Head|Tail],[Head|List]):-
        not(member(Head,Tail)),
        remove_dups(Tail,List).

%=========================================================
% Creates a list of targets assocatied with a policy
%
policy_target_list([Label|[Results]]):-
        policy_targets(Label,Targets),
        create_target_list(Label,Targets,Target_list),
        remove_dups(Target_list,Unique_Target_List),
        flatten(Unique_Target_List, Results).

%=========================================================
% Given a label to identify a policy, creates
% a list of target traffic classes that the
% policy applies to.
%
create_target_list(_,[],[]).
```

```
create_target_list(Label,[_,'!=',_|Tail],Results_of_Tail):-
        policy_action(Label,'deny'),
        create_target_list(Label,Tail,Results_of_Tail).

create_target_list(Label,[_,'*',_|Tail],Results_of_Tail):-
        policy_action(Label,_),
        create_target_list(Label,Tail,Results_of_Tail).

create_target_list(Label,[Class,'==',Target_List|Tail],
        [[Results_of_expand]|Results_of_Tail]):-
        policy_action(Label,Action),
        remove_dups(Target_List, Unique_Target_List),
        expand_value_list(Label,Action,Class,'==',
        Unique_Target_List,Results_of_expand),
        create_target_list(Label, Tail,Results_of_Tail).

%=======================================================
% Creates a list of conditions that must be met
% in order for the policy to be executed.
%
list_of_conditions(Policy,_,[Policy,permit,Attribute,Op,Value]):-
        condition_path(Policy,Attribute,Op,Value),
      not(Op = '!='),
        not(type(Attribute,_,_)).

list_of_conditions(Policy,_,[Policy,permit,Attribute,'!=',Value]):-
        condition_path(Policy,Attribute,'!=',Value),
        not(type(Attribute,_,_)).

%=======================================================
% Creates a list conditions that are composed
% of user defined types.
%
list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Value]):-
        condition_path(Policy,Type,'==',Value).

list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Type_Element]):-
        condition_path(Policy,Type,'!=',_),
        type(Type,Type_Element,_),
        setof(Values,condition_path(Policy,Type,'!=',Values),Value_Set),
        not(member(Type_Element,Value_Set)).

list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Element2]):-
        condition_path(Policy,Type,'<=',Element),
        type(Type,Element,Value),
```

```
        type(Type,Element2,Value2),
        Value2 =< Value.

list_of_type_conditions(Policy,_,[Policy,permit,Type,'==',Element2]):-
        condition_path(Policy,Type,'>=',Element),
        type(Type,Element,Value),
        type(Type,Element2,Value2),
        Value2 >= Value.

%=========================================================
% Prints the conditions that must be meet to
% execute the policy
%
output_conditions([]):-fail.

output_conditions([[Policy,Action,Attribute,Operator,Value]|Tail]):-
        not(Attribute = 'BW'),
        write('policy_condition('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        write(Action),
        write(','),
        term_to_atom(Attribute,A_Attribute),write(A_Attribute),
        write(','),
        term_to_atom(Operator,A_Operator),write(A_Operator),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_conditions(Tail).

output_conditions([[Policy,Action,Attribute,Operator,Value]|Tail]):-
        Attribute = 'BW',
        write('policy_condition('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        write(Action),
        write(','),
        term_to_atom(Attribute,A_Attribute),write(A_Attribute),
        write(','),
        term_to_atom(Operator,A_Operator),write(A_Operator),
        write(','),
        convert_bw(Value,New_Value),
        write(New_Value),
        write(').'),nl,
        output_conditions(Tail).
```

```
%=============================================================
% Prints if there is no conditions that must be meet to
% execute the policy
%
output_no_conditions([]):-fail.

output_no_conditions([Policy|Tail]):-
        write('no_conditions('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(').'),nl,
        output_no_conditions(Tail).

%=============================================================
% When no conditions are associated with a
% policy, makes note of it.
%
print_no_conditions:-
        setof(Policy,no_conditions(Policy),No_Conditions),
        output_no_conditions(No_Conditions),
        true.

%=============================================================
% Prints out the fact if implicit denies are
% to applied to conflict detection when both
% policies are created by the same user
%
print_implicit_deny:-
        user_implicit_deny(Option),
        write('user_implicit_deny('),
        write(Option),
        write(').'),nl,
        true.
%=============================================================
% Prints out a user defined type
% Helper to "print_types"
%
output_types([]):-fail.

output_types([[Type,Element,Value]|Tail]):-
        write('type('),
        term_to_atom(Type,A_Type),write(A_Type),
        write(','),
        term_to_atom(Element,A_Element),write(A_Element),
        write(','),
        write(Value),
        write(').'),nl,
```

```
        output_types(Tail).

%=============================================================
% Prints out all user defined types
%
print_types:-
        setof([Type,Element,Value],type(Type,Element,Value),Types),
        output_types(Types),
        true.

%=============================================================
% For each policy, prints out the target classes
% effected by it.
%
print_target_all:-
        setof(Policy,target(Policy,_),Policies),
        output_target_all(Policies),
        true.

%=============================================================
% Prints out the class of traffic effected by a policy
%
output_target_all([]):-fail.

output_target_all([Policy|Tail]):-
        target(Policy,Value),
        write('target('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_target_all(Tail).

%=============================================================
% Prints out facts about the creator/owner of
% each policy to be applied to the network.
%
print_policy_owner:-
        setof(Policy,policy_owner(Policy,_),Policies),
        output_policy_owner(Policies),
        true.

%=============================================================
% Output the owner for a policy
%
output_policy_owner([]):-fail.
```

```
output_policy_owner([Policy|Tail]):-
        policy_owner(Policy,Value),
        write('policy_owner('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_policy_owner(Tail).

%=========================================================
% Prints out all the nodes of the network.
%
print_nodes:-
        setof(Label,node_label(Label,_),Labels),
        output_nodes(Labels),
        true.

%=========================================================
% Output a fact for each node in the network
% These facts are used in the conflict
% decision process.
%
output_nodes([]):-fail.

output_nodes([Label|Tail]):-
        node_label(Label,Value),
        write('node_label('),
        term_to_atom(Label,A_Label),write(A_Label),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_nodes(Tail).

%=========================================================
% Prints out all the "messages" assciated with a
% link.
%
print_path_messages:-
        setof(Link,path_message(Link,_),Links),
        output_path_messages(Links),
        true.
```

```
%=========================================================
% Prints the messages associated with a policy
%
print_policy_messages:-
        setof(Policy,policy_message(Policy,_),Policies),
        output_policy_messages(Policies),
        true.
%=========================================================
% Outputs the "message" associated with a path
%
output_path_messages([]):-fail.

output_path_messages([Link|Tail]):-
        path_message(Link,Message),
        write('path_message('),
        term_to_atom(Link,A_Link),write(A_Link),
        write(','),
        term_to_atom(Message,A_Message),write(A_Message),
        write(').'),nl,
        output_path_messages(Tail).
%=========================================================
% Outputs the "message" associated with a policy
%
output_policy_messages([]):-fail.

output_policy_messages([Policy|Tail]):-
        policy_message(Policy,Message),
        write('policy_message('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Message,A_Message),write(A_Message),
        write(').'),nl,
        output_policy_messages(Tail).

%=========================================================
% Prints out all the links of the network
%
print_links:-
        findall(Link,link(Link,_,_),Links),!,
        remove_dups(Links, Links_nodup),!,
        output_links(Links_nodup),!,
        true.
```

```
%=========================================================
% Output a link of the network
%
output_links([]):-fail.

output_links([Link|Tail]):-
        link(Link,Src,Dst),!,
        term_to_atom(Src,Src_atom),
        term_to_atom(Dst,Dst_atom),
        term_to_atom(Link,Link_atom),
        write('link('),
        write(Link_atom),
        write(','),
        write(Src_atom),
        write(','),
        write(Dst_atom),
        write(').'),nl,
        write('link('),
        write(Link_atom),
        write(','),
        write(Dst_atom),
        write(','),
        write(Src_atom),
        write(').'),nl,
        output_links(Tail),!,fail.

%=========================================================
% Outputs a user that is allowed to create
% policies.
%
output_users([]):-fail.

output_users([User|Tail]):-
        user(User,Level),
        write('user('),
        term_to_atom(User,A_User),write(A_User),
        write(','),
        term_to_atom(Level,A_Level),write(A_Level),
        write(').'),nl,
        output_users(Tail).
%=========================================================
% Outputs all the actions associated with each policy
%
output_actions([]):-fail.

output_actions([Policy|Tail]):-
```

```
        policy_action(Policy,Action),
        write('policy_action('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Action,A_Action),write(A_Action),
        write(').'),nl,
        output_actions(Tail).

%=========================================================
% Prints all the users that are allowed to
% create policies.
%
print_users:-
        setof(User,user(User,_), User_list),
        output_users(User_list),
        true.

%=========================================================
% Outputs all the actions associated with each policy
%
print_actions:-
        setof(Policy,policy_action(Policy,_), Action_list),
        output_actions(Action_list),
        true.

%=========================================================
% Outputs the parameters associated with a path
% this includes bandwidth.
%
output_params([]):-fail.

output_params([Path|Tail]):-
        path_param(Path,Att,Op,Value,Unit),
        write('path_param('),
        term_to_atom(Path,A_Path),write(A_Path),
        write(','),
        term_to_atom(Att,A_Att),write(A_Att),
        write(','),
        term_to_atom(Op,A_Op),write(A_Op),
        write(','),
        convert_unit(Value,Unit,New_Value),
        term_to_atom(New_Value,A_Value),write(A_Value),
        write(','),
        term_to_atom('MBPS',A_Unit),write(A_Unit),
        write(').'),nl,
        output_params(Tail).
```

```
%=========================================================
% Prints out all the paramaters associated with
% each path
%
print_path_params:-
        setof(Path, path_param(Path,_,_,_,_),Paths),
        output_params(Paths),
        true.

%=========================================================
% Prints out the conditions of all policies
%
print_condition_list:-
        setof(Condition_Set,list_of_conditions(_,_,Condition_Set), Conditions),
        output_conditions(Conditions),
        true.

%=========================================================
% Prints out the user defined types involved in the
% conditions of all policies
%
print_condition_list:-
        setof(Condition_Set,list_of_type_conditions(_,_,Condition_Set),
        Conditions), output_conditions(Conditions), true.

%=========================================================
% Converts Bandwidth unit to a uniform Mbps for
% comparison reasons
%
convert_bw([Value,Unit],New_Value):-
        convert_unit(Value,Unit,New_Value).

%=========================================================
convert_unit(Old_Value,'MBPS',Old_Value).

convert_unit(Old_Value,'GBPS',New_Value):-
        New_Value is Old_Value * 1024.

convert_unit(Old_Value,'KBPS',New_Value):-
        New_Value is Old_Value / 1024.

convert_unit(Old_Value,'BPS',New_Value):-
        New_Value is (Old_Value / 1024)/1024.
```

```
%=========================================================
% True if there is an link from X->Y or Y->X (undirected link)
%
adjacent(X, Y) :- link(_,X,Y); link(_,Y,X).

%=========================================================
% Concats two lists together
%
conc([],L,L).
conc( [X | L1], L2, [X | L3] ) :- conc(L1,L2,L3).

%=========================================================
% Is the list of size 1?
%
eq1([_|[]]).
```

```
% Stage2 is the second step in determining policy conflicts.
% Steps:  * Opens "ppl3.txt" file for policy conflict information
%         * Prints all paths in the next that are associated with
%           policies being applied to the network.
%         * Copies all other neccesary facts from stage two to
%           the file that will be used in third and final stage three.
%         * Closes the output file
%
% Authors: Neil ROWE, Gary STONE, Ahmet GUVEN
%
stage2:-write('Stage 2 started'),nl,
        open('ppl3.txt',write,output),
        set_output(output),

        % Print the paths associated with policies.
        %
        not(all_policy_paths),

        % Print all necessary facts for use in stage three
        listing(bic_list),
        not(print_no_conditions),
        not(print_path_params),
        not(print_users),
        print_implicit_deny,
        not(print_types),
        not(print_actions),
        not(print_target_all),
        not(print_policy_owner),
        not(print_nodes),
        not(print_links),
        not(print_path_messages),
        not(print_policy_messages),
        write('condition(_null,_null,_null,_null,_null,_null).'),nl,

        set_output(user_output),
        close(output),
        write('Stage 2 complete'),nl,nl.

%=======================================================
% Prints all paths associated with policies
%
all_policy_paths:-
      policy_path(Pol_Label,Pol_Path),
      setof(Pol_Paths,collect_paths(Pol_Label,Pol_Paths),Policy_Paths),
      not(print_path_list(Policy_Paths)),
         print_conditions(Policy_Paths).
```

```
%=========================================================
% Helps to print path associated with policies.
%
collect_paths(Pol_Label,Pol_path):-policy_path(Pol_Label,Policy_Path),
        Pol_path=[Pol_Label|Policy_Path].

%=========================================================
% True if there is an link from X->Y or Y->X (undirected link)
%
adjacent(X, Y) :- link(_,X,Y); link(_,Y,X).

%=========================================================
% print_path_list:
% Output a list of lists, where each
% list is on a line by itself and
% all the atoms are quoted.
%
print_path_list([]):- fail.

print_path_list([[Policy_Label|Path] | Tail]) :-
        policy_target(Policy_Label,Policy_Targets),
        not(eq1([Policy_Label|Path])),
        write('path('),
        term_to_atom(Policy_Label,Policy_atom),
        write(Policy_atom),
        write(',['),
        write_path(Path),
        write(',['),
        write_path(Policy_Targets),
        write(').'), nl, not(print_path_list(Tail)),fail.

print_path_list([[Policy_Label|Path] | Tail]) :-
        not(policy_target(Policy_Label,_)),
        not(eq1([Policy_Label|Path])),
        write('path('),
        term_to_atom(Policy_Label,Policy_atom),
        write(Policy_atom),
        write(',['),
        write_path(Path),
        write(',[]).'), nl,not(print_path_list(Tail)),fail.
```

```
%=========================================================
% Prints the conditions associated with every policy
%
print_conditions([]):- fail.

print_conditions([[Policy_Label|Path] | Tail]) :-
        policy_condition(Policy_Label,Action,Att,Op,Value),
        not(eq1([Policy_Label|Path])),
        write('condition('),
        term_to_atom(Policy_Label,Policy_atom),write(Policy_atom),
        write(',['),
        write_path(Path),
        write(','),
        term_to_atom(Action,Action_atom),write(Action_atom),
        write(','),
        term_to_atom(Att,Att_atom),write(Att_atom),
        write(','),
        term_to_atom(Op,Op_atom),write(Op_atom),
        write(','),
        term_to_atom(Value,Value_atom),write(Value_atom),
        write(').'), nl, print_path_list(Tail).

print_conditions([[Policy_Label|Path] | Tail]) :-
        not(policy_condition(Policy_Label,_,_,_,_)),
        not(eq1([Policy_Label|Path])),
        print_path_list(Tail).

%=========================================================
% Helper function used by print_path_list to output
% a list with all the atoms quoted.
%
write_path([]):- write(']').

write_path([X|[]]):- term_to_atom(X,X_atom),
                     write(X_atom),
                     write(']'),!,true.

write_path([X|Tail]):- term_to_atom(X,X_atom),
                       write(X_atom),
                       write(','),
                       write_path(Tail).
```

```
%=========================================================
% Is the list of size 1?
%
eq1([_|[]]).

%=========================================================
% Prints a user defined type
%
output_types([]):-fail.

output_types([[Type,Element,Value]|Tail]):-
        write('type('),
        term_to_atom(Type,A_Type),write(A_Type),
        write(','),
        term_to_atom(Element,A_Element),write(A_Element),
        write(','),
        write(Value),
        write(').'),nl,
        output_types(Tail).

%=========================================================
% Prints all the user defined types
%
print_types:-
        setof([Type,Element,Value],type(Type,Element,Value),Types),
        output_types(Types),
        true.
%=========================================================
% Prints a policy that has no conditions
% associated with it.
%
output_no_conditions([]):-fail.

output_no_conditions([Policy|Tail]):-
        write('no_conditions('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(').'),nl,
        output_no_conditions(Tail).

%=========================================================
% Prints all the policies that have no conditions
% associated with them.
%
print_no_conditions:-
        setof(Policy,no_conditions(Policy),No_Conditions),
        not(No_Conditions = []),
```

```
        output_no_conditions(No_Conditions),
        true.
%=========================================================
% Prints the owners of all the policies
%
print_policy_owner:-
        setof(Policy,policy_owner(Policy,_),Policies),
        output_policy_owner(Policies),
        true.

%=========================================================
% Prints the owner of a policy
%
output_policy_owner([]):-fail.

output_policy_owner([Policy|Tail]):-
        policy_owner(Policy,Value),
        write('policy_owner('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_policy_owner(Tail).

%=========================================================
% Prints the nodes of the network.
%
print_nodes:-
        setof(Label,node_label(Label,_),Labels),
        output_nodes(Labels),
        true.

%=========================================================
% Prints out a node of the network
%
output_nodes([]):-fail.

output_nodes([Label|Tail]):-
        node_label(Label,Value),
        write('node_label('),
        term_to_atom(Label,A_Label),write(A_Label),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_nodes(Tail).
```

```
%=========================================================
% Prints out all the links of the network
%
print_links:-
        findall(Link,link(Link,_,_),Links),!,
        remove_dups(Links, Links_nodup),!,
        output_links(Links_nodup),!,
        true.

%=========================================================
% Prints out a link in the network.
%
output_links([]):-fail.

output_links([Link|Tail]):-
        link(Link,Src,Dst),!,
        term_to_atom(Src,Src_atom),
        term_to_atom(Dst,Dst_atom),
        term_to_atom(Link,Link_atom),
        write('link('),
        write(Link_atom),
        write(','),
        write(Src_atom),
        write(','),
        write(Dst_atom),
        write(').'),nl,
        write('link('),
        write(Link_atom),
        write(','),
        write(Dst_atom),
        write(','),
        write(Src_atom),
        write(').'),nl,
        output_links(Tail),!,fail.

%=========================================================
% Prints out messages associated with each path
%
print_path_messages:-
        setof(Link,path_message(Link,_),Links),
        output_path_messages(Links),true,!.

%=========================================================
% Prints the messages associted with each policy
%
print_policy_messages:-
```

```
        setof(Policy,policy_message(Policy,_),Policies),
        output_policy_messages(Policies),true,!.

%=======================================================
% Prints the messages associated with a path
%
output_path_messages([]):- fail.

output_path_messages([Link|Tail]):-
        path_message(Link,Message),
        write('path_message('),
        term_to_atom(Link,A_Link),write(A_Link),
        write(','),
        term_to_atom(Message,A_Message),write(A_Message),
        write(').'),nl,
        output_path_messages(Tail),!.
%=======================================================
% Prints the messages assocaited with a path
%
output_policy_messages([]):- fail.

output_policy_messages([Policy|Tail]):-
        policy_message(Policy,Message),
        write('policy_message('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Message,A_Message),write(A_Message),
        write(').'),nl,
        output_policy_messages(Tail),!.

%=======================================================
% Prints out the users who can created policies
%
output_users([]):- fail.

output_users([User|Tail]):-
        user(User,Level),
        write('user('),
        term_to_atom(User,A_User),write(A_User),
        write(','),
        write(Level),
        write(').'),nl,
        output_users(Tail).
```

```
%=========================================================
% Prints the actions associated with policy
%
output_actions([]):-fail.

output_actions([Policy|Tail]):-
        policy_action(Policy,Action),
        write('policy_action('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Action,A_Action),write(A_Action),
        write(').'),nl,
        output_actions(Tail).

%=========================================================
% Prints the paramaters associated with a path
%
output_params([]):-fail.

output_params([Path|Tail]):-
        path_param(Path,Att,Op,Value,Unit),
        write('path_param('),
        term_to_atom(Path,A_Path),write(A_Path),
        write(','),
        term_to_atom(Att,A_Att),write(A_Att),
        write(','),
        term_to_atom(Op,A_Op),write(A_Op),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(','),
        term_to_atom(Unit,A_Unit),write(A_Unit),
        write(').'),nl,
        output_params(Tail).

%=========================================================
% Prints the parameters of each path
%
print_path_params:-
        setof(Path, path_param(Path,_,_,_,_),Paths),
        output_params(Paths),
        true.
%=========================================================
% Print the flag identifing whether implicit
% denies are to be ignored between policies
% created by the same user
%
```

```
print_implicit_deny:-
        user_implicit_deny(Option),
        write('user_implicit_deny('),
        write(Option),
        write(').'),nl,
        true.

%=========================================================
% Prints all the user who can create a policy
%
print_users:-
        setof(User,user(User,_), User_list),
        output_users(User_list),
        true.

%=========================================================
% Prints out all the actions for each policy
%
print_actions:-
        setof(Policy,policy_action(Policy,_), Action_list),
        output_actions(Action_list),
        true.

%=========================================================
% Prints the targets for each policy
%
print_target_all:-
        setof(Policy,target(Policy,_),Policies),
        output_target_all(Policies),
        true.

%=========================================================
% Prints the target for a policy
%
output_target_all([]):-fail.

output_target_all([Policy|Tail]):-
        target(Policy,Value),
        write('target('),
        term_to_atom(Policy,A_Policy),write(A_Policy),
        write(','),
        term_to_atom(Value,A_Value),write(A_Value),
        write(').'),nl,
        output_target_all(Tail).
```

```
%=======================================================
% Removes duplicate items from a list
%
remove_dups([],[]).

remove_dups([Head|Tail],List):-
        member(Head,Tail),
        remove_dups(Tail,List).

remove_dups([Head|Tail],[Head|List]):-
        not(member(Head,Tail)),
        remove_dups(Tail,List).
```

```
% Stage3 is the final step in determining policy conflicts.
% Steps: * Open "scan_out.txt" file for policy conflict information
%          * Check policies first overlapping conditions,
%              then overlapping targets and than overlapping physical
%              paths to detect conflicts
%          * Determine policies in conflict
%          * Print the policy conflicts that could not be resolved
%          * Print the policy conflicts that COULD be resolved
%          * Print out the policies that contain "message" conflicts
%          * Close the output file
%
% Authors: Neil ROWE, Gary STONE, Ahmet GUVEN
%

:- dynamic agenda/3, usedstate/3, compared/6,
        already_expanded/3,overlapping/2.
:- style_check(-singleton).
:- unknown(_, fail).

stage3:- open('scan_out5.txt',write,output),
          set_output(output),

         %remove duplicate path definitions from the memory
         clean_memory,

         % Check policies first overlapping conditions,
         % then overlapping targets and than overlapping physical
         % paths to detect conflicts.
         %
         find_conflicting_policies(Conflicting_Policies),

         % Print out policy conflicts that can not be
         % resolved using the "Id" of the creator
         %
         write('                Print Unresolved Conflicts'),nl,
         write('                ======================='),nl,
         print_unresolved_conflict_list(Conflicting_Policies),

         % Print out the policy conflicts than CAN be resolved
         % using the "Id" of the policy creator
         %
         nl,nl,nl,write('                  Print Resolved Conflicts'),nl,
         write('                ======================='),nl,
         print_resolved_conflict_list(Conflicting_Policies),

         % Print out the policies that require "message" support on a
```

```
        % path, but all the links of the path do not support that "message"
        %
        nl,nl,nl,write('                Message Conflicts'),nl,
        write('                ================'),nl,nl,
        not(print_message_conflict_list),

        %
        set_output(user_output),
        close(output),
        write('Stage 3 complete'),nl,nl.

%=========================================================
% Return set of all policies that conflict to eachother.
% First, policy conditions are compared for conflicts
% After that policy targets are compared. Only
% policies which have conflicting targets and
% conditions are checked for physical overlaps.
%
find_conflicting_policies(Conflicting_Policies):-
        retract_all(compared(_,_,_,_,_,_)), retract_all(overlapping(_,_)),
        retract_all(already_expanded(_,_,_)).

find_conflicting_policies(Conflicting_Policies):-
        setof(Overlaps, permit_conflicts(Overlaps),Conflicting_Policies).

find_conflicting_policies(Conflicting_Policies):-
        not(setof(Overlaps, permit_conflicts(Overlaps),_)).

%=========================================================
% Compares two policy paths and finds physical overlaps.
% Compared policy paths are recorded to memory. If the
% policy paths are compared before, rule returns whether
% they have overlap or not without seraching. This is an
% important issue for minimizing the number of policy
% comparations.
%
find_physical_overlaps([Policy1,Path1,Target1,Policy2,Path2,Target2]):-
        path(Policy1,Path1,Target1),
        path(Policy2,Path2,Target2),not(Policy1=@=Policy2),
        ifthen((compared(_,Path1,_,_,Path2,_),not(overlapping(Path1,Path2))),
        (fail,!)),
        not compared(Policy1,Path1,Target1,Policy2,Path2,Target2),
        assertz(compared(Policy1,Path1,Target1,Policy2,Path2,Target2)),
        ((overlapping(Path1,Path2);overlapping(Path2,Path1));
        check_physical_overlap(Path1,Path2),
        assertz(overlapping(Path1,Path2)) ).
```

```
%=======================================================
% Checks each policy pair for overlapping targets and conditions.
% If the targets conditions do overlap, then check checks
% for the physical path overlaps.
%
permit_conflicts([Policy1,Path1,Target1,Policy2,Path2,Target2,Conflicts]):-
        path(Policy1,Path1,Target1),
        path(Policy2,Path2,Target2),
        not(Policy1 = Policy2),
        conditional_overlap(Policy1,Policy2,_,No_Overlap),
        No_Overlap = [],
        setof(Conflict,conflict_permit_targets(Policy1,Policy2,Target2,
        Target1,Conflict),Conflicts),
        not(Conflicts= [[]]),
        find_physical_overlaps([Policy1,Path1,Target1,Policy2,Path2,Target2]).

permit_conflicts([Policy1,Path1,Target1,Policy2,Path2,
        Target2,[[Policy1,Value]]]):-
        path(Policy1,Path1,Target1),
        path(Policy2,Path2,Target2),
        not(Policy1 = Policy2),
        conditional_overlap(Policy1,Policy2,_,No_Overlap),
        No_Overlap = [],
        (
        (target(Policy1, permit_all),
        policy_action(Policy2,deny));
        (target(Policy1, deny_all),
        policy_action(Policy2,permit))
        ),
        target(Policy1,Value),find_physical_overlaps([Policy1,Path1,Target1,
        Policy2,Path2,Target2]).

permit_conflicts( [Policy1,Path1,Target1,Policy2,Path2,Target2,
        [[Policy2,Value]]]):-
        path(Policy1,Path1,Target1),
        path(Policy2,Path2,Target2),
        not(Policy1 = Policy2),
        conditional_overlap(Policy1,Policy2,_,No_Overlap),
        No_Overlap = [],
        (
        (target(Policy2, permit_all),
        policy_action(Policy1,deny));
        (target(Policy2, deny_all),
        policy_action(Policy1,permit))),
        target(Policy2,Value),find_physical_overlaps([Policy1,Path1,Target1,
        Policy2,Path2,Target2]).
```

```
permit_conflicts(_).


%=============================================================
% Compares a target class element for conflicts with all
% the target elements of a second policy
%
conflict_permit_targets(_,_,[],_,[]).

conflict_permit_targets(Policy1,Policy2,[A1,C,_,V|_],Targets,Results):-
        conflict_permit_target(Policy1,Policy2,[A1,C,V],Targets,Results),
        not(empty(Results)).

conflict_permit_targets(Policy1,Policy2,[_,_,_,_|Tail],Targets,Results):-
        conflict_permit_targets(Policy1,Policy2,Tail,Targets,Results),
        not(empty(Results)).

%=============================================================
% Compares the target class and action between two
% two target elements.
%
conflict_permit_target(Policy1,Policy2,['permit',C,V],[],[C,V]):-
        policy_owner(Policy1, Creator1),
        policy_owner(Policy2, Creator2),
        Creator1 = Creator2,
        user_implicit_deny('no'),
        fail.

conflict_permit_target(_,_,['permit',C,V],[],[C,V]):-
        user_implicit_deny('yes').

conflict_permit_target(_,_,['deny',_,_],[],[]).

conflict_permit_target(_,_,[],_,[]).

conflict_permit_target(_,_,[A1,C,Value],[A2,C,_,Value|_],[C,Value]):-
        not(A1 = A2),!.

conflict_permit_target(_,_,[A,C,Value],[A,C,_,Value|_],[]):-!.

conflict_permit_target(Policy1,Policy2,[A1,C1,Value1],[_,_,_,_|Tail],Results):-
        conflict_permit_target(Policy1,Policy2,[A1,C1,Value1],Tail,Results).
```

```
%=======================================================
% Determines if a conditional overlap exists by testing each
% type of condition.
%
conditional_overlap(P1,P2,['priority'|Over],No_Over):-
        priority_overlap(P1,P2),
        conditional_overlap1(P1,P2,Over,No_Over),!.

conditional_overlap(P1,P2,Over,['priority'|No_Over]):-
        not(priority_overlap(P1,P2)),
        conditional_overlap1(P1,P2,Over,No_Over),!.

conditional_overlap1(P1,P2,['time'|Over],No_Over):-
        time_overlap(P1,P2),
        conditional_overlap2(P1,P2,Over,No_Over),!.

conditional_overlap1(P1,P2,Over,['time'|No_Over]):-
        not(time_overlap(P1,P2)),
        conditional_overlap2(P1,P2,Over,No_Over),!.

conditional_overlap2(P1,P2,['hopcount'|Over],No_Over):-
        hopcount_overlap(P1,P2),
        conditional_overlap3(P1,P2,Over,No_Over),!.

conditional_overlap2(P1,P2,Over,['hopcount'|No_Over]):-
        not(hopcount_overlap(P1,P2)),
        conditional_overlap3(P1,P2,Over,No_Over),!.

conditional_overlap3(P1,P2,['bandwidth'|Over],No_Over):-
        bw_overlap(P1,P2),
        conditional_overlap4(P1,P2,Over,No_Over),!.

conditional_overlap3(P1,P2,Over,['bandwidth'|No_Over]):-
        not(bw_overlap(P1,P2)),
        conditional_overlap4(P1,P2,Over,No_Over),!.

conditional_overlap4(P1,P2,['user'|Over],No_Over):-
        user_overlap(P1,P2),
        conditional_overlap5(P1,P2,Over,No_Over),!.

conditional_overlap4(P1,P2,Over,['user'|No_Over]):-
        not(user_overlap(P1,P2)),
        conditional_overlap5(P1,P2,Over,No_Over),!.

conditional_overlap5(P1,P2,['host'|Over],No_Over):-
        host_overlap(P1,P2),
```

```
        conditional_overlap6(P1,P2,Over,No_Over),!.

conditional_overlap5(P1,P2,Over,['host'|No_Over]):-
        not(host_overlap(P1,P2)),
        conditional_overlap6(P1,P2,Over,No_Over),!.

conditional_overlap6(P1,P2,['type'|Over],No_Over):-
        not(overlap_types(P1,P2,_)),
        conditional_overlap7(P1,P2,Over,No_Over),!.

conditional_overlap6(P1,P2,Over,['type'|No_Over]):-
        overlap_types(P1,P2,_),
        conditional_overlap7(P1,P2,Over,No_Over),!.

conditional_overlap7(_,_,[],[]).

%=============================================
% Determines if there is an overlap between
% user defined types.
%
overlap_types(Policy1, Policy2, Type):-
        type(Type,_,_),
        setof(Element,condition(Policy1,_,_,Type,_,Element),Elements1),
        setof(Element,condition(Policy2,_,_,Type,_,Element),Elements2),
        not(Elements1 = []),
        not(Elements2 = []),
        intersection(Elements1,Elements2,I),
        I = [].

%=============================================
% Determines if a priority overlap exists
%
priority_overlap(Policy1,Policy1).

priority_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        (not(setof([Op1,Value1],condition(Policy1,_,_,'priority',Op1,Value1),_));
         not(setof([Op2,Value2],condition(Policy2,_,_,'priority',Op2,Value2),_))).

priority_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,Value1],condition(Policy1,_,_,'priority',Op1,Value1),Values1),
        setof([Op2,Value2],condition(Policy2,_,_,'priority',Op2,Value2),Values2),
        overlap(Values1,Values2).
```

```
%=============================================
% Determines if a hopcount overlap exists
%
hopcount_overlap(Policy1,Policy1).

hopcount_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        (not(setof([Op1,Value1],
        condition(Policy1,_,_,'hopcount',Op1,Value1),_));
        not(setof([Op2,Value2],
        condition(Policy2,_,_,'hopcount',Op2,Value2),_))).

hopcount_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,Value1],
        condition(Policy1,_,_,'hopcount',Op1,Value1),Values1),
        setof([Op2,Value2],
        condition(Policy2,_,_,'hopcount',Op2,Value2),Values2),
        overlap(Values1,Values2).

%=============================================
% Determines if a bandwidth overlap exists
%
bw_overlap(Policy1,Policy1).

bw_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        (not(setof([Op1,Value1],condition(Policy1,_,_,'BW',Op1,Value1),_));
         not(setof([Op2,Value2],condition(Policy2,_,_,'BW',Op2,Value2),_))).

bw_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,Value1],condition(Policy1,_,_,'BW',Op1,Value1),Values1),
        setof([Op2,Value2],condition(Policy2,_,_,'BW',Op2,Value2),Values2),
        overlap(Values1,Values2).

%=============================================
% Determines if a time overlap exists
%
time_overlap(Policy1,Policy1).

time_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        (not(setof([Op1,Time1],condition(Policy1,_,_,'time',Op1,Time1),_));
         not(setof([Op2,Time2],condition(Policy2,_,_,'time',Op2,Time2),_))).
```

```
time_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,Time1],condition(Policy1,_,_,'time',Op1,Time1),Times1),
        setof([Op2,Time2],condition(Policy2,_,_,'time',Op2,Time2),Times2),
        overlap(Times1,Times2).

%=============================================
% An overlap exists if ALL elements of a condition
% overlap. The overlap rule uses the operator and value
% of two conditional elements to determine if overlaps
% exist
%
overlap([],[_|_]).

overlap([_|_],[]).

overlap([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2]):-
        not([[Op2,Value2]|Tail2] = []),
        not([[Op1,Value1]|Tail1] = []),
        overlap1(Value1,Op1,Value2,Op2),
        overlap(Tail1,[[Op2,Value2]|Tail2]),
        overlap([[Op1,Value1]|Tail1],Tail2).

%=============================================
% Helper rule to "overlap" above.
% Detemines if an overlap exists between
% two conditional elements.
%
overlap1(Value1,_,Value1,_).

overlap1(_,'>=',_,'>=').

overlap1(Value1,'>=',Value2,'<='):-
        Value1 < Value2.

overlap1(_,'<=',_,'<='):-true.

overlap1(Value1,'<=',Value2,'>='):-
        Value1 > Value2.

%=============================================
% Determines if there is an overlap between
% users. Each user in the conditional element
% list is checked, and if any user
% overlap exists, then the policy contains a
% user overlap.
```

```
%
user_overlap(Policy1,Policy1).

user_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        (not(setof([Op1,User1],condition(Policy1,_,_,'user_id',Op1,User1),_));
         not(setof([Op2,User2],condition(Policy2,_,_,'user_id',Op2,User2),_))).

user_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,User1],condition(Policy1,_,_,'user_id',Op1,User1),Users1),
        setof([Op2,User2],condition(Policy2,_,_,'user_id',Op2,User2),Users2),
        user_overlap1(Users1,Users2,Results),
        flatten(Results,Flat_Results),
        not(member('conflict',Flat_Results)),
        member('overlap',Flat_Results),
        true.

user_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,User1],condition(Policy1,_,_,'user_id',Op1,User1),Users1),
        setof([Op2,User2],condition(Policy2,_,_,'user_id',Op2,User2),Users2),
        user_overlap1(Users1,Users2,Results),
        flatten(Results,Flat_Results),
        member('conflict',Flat_Results),
        fail.

%=============================================
% Helper rule to "user_overlap" above.
% Determines if user element lists overlap
% with at least one common user
%
user_overlap1([],[_|_],[]).

user_overlap1([_|_],[],[]).

user_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],
        [Result1,Result2,Result3]):-
        not([[Op2,Value2]|Tail2] = []),
        not([[Op1,Value1]|Tail1] = []),
        user_overlap2(Value1,Op1,Value2,Op2,Result1),
        user_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
        user_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

user_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result2,Result3]):-
        not([[Op2,Value2]|Tail2] = []),
```

```
        not([[Op1,Value1]|Tail1] = []),
        not(user_overlap2(Value1,Op1,Value2,Op2,_)),
        user_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
        user_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

%=============================================
% Helper rule to "user_overlap1" above
% Determines if two user values overlap
% Can return "No overlap", "Overlap", or "Conflict"
%
user_overlap2(Value,Op,Value,Op,'overlap').

user_overlap2(Value1,'==',Value2,'==','nooverlap'):-
        not(Value1 = Value2).

user_overlap2(Value1,'==',Value2,'!=','overlap'):-
        not(Value1 = Value2).

user_overlap2(Value1,'==',Value1,'!=','conflict').

% This makes an assumption that there are many/infinite users
% If there were limited user, then maybe they should be a "type" instead.
%
user_overlap2(_,'!=',_,'!=','overlap').

%=============================================
% Determines if there is an overlap in host identifiers
%
host_overlap(Policy1,Policy1).

host_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        (not(setof([Op1,Add1],condition(Policy1,_,_,'host_id',Op1,Add1),_));
         not(setof([Op2,Add2],condition(Policy2,_,_,'host_id',Op2,Add2),_))).

host_overlap(Policy1,Policy2):-
        not(Policy1 = Policy2),
        setof([Op1,Add1],condition(Policy1,_,_,'host_id',Op1,Add1),Addresses1),
        setof([Op2,Add2],condition(Policy2,_,_,'host_id',Op2,Add2),Addresses2),
        host_overlap1(Addresses1,Addresses2,Results),
        flatten(Results,Flat_Results),
        not(member('conflict',Flat_Results)),
        member('overlap',Flat_Results),
        true.

host_overlap(Policy1,Policy2):-
```

```
        not(Policy1 = Policy2),
        setof([Op1,Add1],condition(Policy1,_,_,'host_id',Op1,Add1),Addresses1),
        setof([Op2,Add2],condition(Policy2,_,_,'host_id',Op2,Add2),Addresses2),
        host_overlap1(Addresses1,Addresses2,Results),
        flatten(Results,Flat_Results),
        member('conflict',Flat_Results),
        fail.

%=============================================
% Helper rule to "host_overlap" above
% Progresses through the list of host addresses from
% two policies looking for an overlap
%
host_overlap1([],[_|_],[]).

host_overlap1([_|_],[],[]).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],
        [Result1,Result2,Result3]):-
        not([[Op2,Value2]|Tail2] = []),
        not([[Op1,Value1]|Tail1] = []),
        not(address_overlap(Value1,Value2)),
        host_overlap2(Op1,'X',Op2,'Y',Result1),
        host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
        host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],
        [Result1,Result2,Result3]):-
        not([[Op2,Value2]|Tail2] = []),
        not([[Op1,Value1]|Tail1] = []),
        address_overlap(Value1,Value2),
        host_overlap2(Op1,'X',Op2,'X',Result1),
        host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
        host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result2,Result3]):-
        not([[Op2,Value2]|Tail2] = []),
        not([[Op1,Value1]|Tail1] = []),
        address_overlap(Value1,Value2),
        not(host_overlap2(Op1,'X',Op2,'X',_)),
        host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
        host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

host_overlap1([[Op1,Value1]|Tail1],[[Op2,Value2]|Tail2],[Result2,Result3]):-
        not([[Op2,Value2]|Tail2] = []),
        not([[Op1,Value1]|Tail1] = []),
```

```
        not(address_overlap(Value1,Value2)),
        not(host_overlap2(Op1,'X',Op2,'Y',_)),
        host_overlap1(Tail1,[[Op2,Value2]|Tail2],Result2),
        host_overlap1([[Op1,Value1]|Tail1],Tail2,Result3).

%=============================================
% Helper rule to "host_overlap1" above
% Determines if two host values overlap using
% the specified comparison operator
%
host_overlap2('==',X,'==',X,'overlap').

host_overlap2('==',X,'==',Y,'nooverlap'):- not(X = Y).

host_overlap2('==',X,'!=',X,'conflict').

host_overlap2('==',X,'!=',Y,'overlap'):- not(X = Y).

host_overlap2('!=',X,'==',X,'conflict').

host_overlap2('!=',X,'==',Y,'overlap'):- not(X = Y).

host_overlap2('!=',X,'!=',X,'overlap').

host_overlap2('!=',X,'!=',Y,'overlap'):- not(X = Y).

%=============================================
% Determines if IP protocol addresses overlap
%

address_overlap([Dot1|Tail1],[Dot2|Tail2]):-
        Dot1 = Dot2,
        address_overlap(Tail1,Tail2).

address_overlap([Dot1|_],[Dot2|_]):-
        Dot1 = '*';
        Dot2 = '*'.

address_overlap([],[]).

%=============================================
% Determines if message specified for a policy path
% is supported by each link that composes the path
%
message_conflict(Policy,[Policy,Results]):-
        path(Policy,_,_),
```

```
        setof(Link, message_conflict_helper(Policy,[_,Link]), Links),
        flatten(Links,Flattened_Links),
        generate_list_pairs(Flattened_Links, List_of_Pairs),
        remove_dups(List_of_Pairs,Results).

message_conflict(Policy,[]):-
        path(Policy,_,_),
        not(setof(Link, message_conflict_helper(Policy,[_,Link]), _)).

%===========================================
% Given a policy, return a list of links that do
% NOT support the needed "messages"
%
message_conflict_helper(Policy,[Policy,Results]):-
        path(Policy,Path,_),
        policy_message(Policy,Message),
        message_supported(Path,Message,Results),
        not(Results = []).
%===========================================
% Create a list of link, message pairs such that
% the pairs returned do NOT support the needed "message"
%
message_supported([],_,[]).

message_supported([Src,Dst],Message,[Link,Message]):-
        link(Link,Src,Dst),
        not(path_message(Link,Message)).

message_supported([Src,Dst],Message,[]):-
        link(Link,Src,Dst),
        path_message(Link,Message).

message_supported([Src,Dst|Tail], Message, Results):-
        link(Link,Src,Dst),
        path_message(Link,Message),
        message_supported([Dst|Tail], Message,Results).

message_supported([Src,Dst|Tail], Message, [Link,Message|Results]):-
        link(Link,Src,Dst),
        not(path_message(Link,Message)),
        message_supported([Dst|Tail], Message,Results).


%===========================================
% Modify a given list of items into a list of
% pairs, taking two elements at time to form
```

```
% the pairs.
%
generate_list_pairs([],[]).

generate_list_pairs([Node,Message|Tail],[[Node,Message]|Tail_Results]):-
        generate_list_pairs(Tail,Tail_Results).

%=========================================================
% Remove duplicate items from a list
%
remove_dups([],[]).

remove_dups([Head|Tail],List):-
        member(Head,Tail),
        remove_dups(Tail,List).

remove_dups([Head|Tail],[Head|List]):-
        not(member(Head,Tail)),
        remove_dups(Tail,List).
%=========================================================
% Concat two lists together
%
conc([],L,L).

conc( [X | L1], L2, [X | L3] ) :- conc(L1,L2,L3).

%=========================================================
% Is a list empty
%
empty([]):-true.

empty([_|_]):-fail.

%=========================================================
% Compares two policy paths for physical overlaps
%

% if a wildcard character is used to represent all paths
% it has overlapping links with any other policy paths
%
check_physical_overlap(*,List).

check_physical_overlap(List,*).
```

```
% deals with cases like {*,A,*}
%
check_physical_overlap([*,A,*],List):-!, member(*,List);expand(List).

check_physical_overlap(List,[*,A,*]):-!, member(*,List);expand(List).

% If two nodes are being compared: if they are
% same, there is an overlap. If they are not,
% there is no overlap
%
check_physical_overlap([A],[B]):-A=@=B,!.

check_physical_overlap([A],[B]):-not(A=@=B),!,fail,!.

% calls 'check_physical_overlap1' and 'check_physical_overlap2'
% to deal with the cases when one of the policy paths a single node
% and the other is one using a wildcard character.
%
check_physical_overlap([A],[C,*,D]):-!,check_physical_overlap1([A],[C,*,D]),!.

check_physical_overlap([C,*,D],[A]):-!,check_physical_overlap2([C,*,D],[A]),!.

% calls 'check_physical_overlap3'and 'check_physical_overlap4'
% to deal with the cases when one of the policy paths a single
% node and the other is one is list of nodes.
%
check_physical_overlap([A],List):-!,check_physical_overlap3([A],List),!.

check_physical_overlap(List,[A]):-!,check_physical_overlap4(List,[A]),!.

% calls and 'check_physical_overlap5' to deal with the cases
% when both of the policy paths use wildcard characters.
%
check_physical_overlap([A,*,B],[C,*,D]):-!,
        check_physical_overlap5([A,*,B],[C,*,D]),!.

%deals with the cases when one of the policy paths is
%a link and the other uses wildcard characters.
%
check_physical_overlap([A,B],[C,*,D]):-!,
        check_physical_overlap6([A,B],[C,*,D]),!.

check_physical_overlap([C,*,D],[A,B]):-!,
        check_physical_overlap7([C,*,D],[A,B]),!.
```

```
%=======================================================
% deals with the cases when one of the policy paths (first one)
% a single node and the other is one using a wildcard character.
% returns overlap if the node is in the same biconnected component
% with any of the wildcard path nodes or the node is the either
% the source or destination node.
%
check_physical_overlap1([A],[C,*,D]):- A=@=C;A=@=D.

% If the node is in the same biconnected group with either of source
% or destination node of wildcard path, there is a overlap
%
check_physical_overlap1([A],[C,*,D]):-bic_list(_,Biconnected_Group),
        member(A,Biconnected_Group),member(C,Biconnected_Group);
        member(A,Biconnected_Group),member(D,Biconnected_Group),!.


% If the node is in a diffent biconnected component than the other
% policy nodes, returns no overlap.
%
check_physical_overlap1([A],[C,*,D]):-bic_list(_,Biconnected_Group1),
        member(C,Biconnected_Group1),
        member(D,Biconnected_Group1),
        not member(A,Biconnected_Group1),!,fail,!.

%If the problem can not be solved with previous cases
%
check_physical_overlap1([A],[C,*,D]):- check_physical_overlap8([A],[C,*,D]).

%=======================================================
% deals with the cases when one of the policy paths (second one)
% a single node and the other is one using a wildcard character.
% returns overlap if the node is in the same biconnected component
% with any of the wildcard path nodes or the node is the either
% the source or destination node.
%
check_physical_overlap2([C,*,D],[A]):- A=@=C;A=@=D.

% If the node is in the same biconnected group with either of source
% or destination node of wildcard path, there is a overlap
%
check_physical_overlap2([C,*,D],[A]):-bic_list(_,Biconnected_Group),
        member(A,Biconnected_Group),member(D,Biconnected_Group);
        member(A,Biconnected_Group),member(C,Biconnected_Group),!.
```

```
% If the node is in a diffent biconnected component than the other
% policy nodes, returns no overlap.
%
check_physical_overlap2([C,*,D],[A]):-bic_list(_,Biconnected_Group1),
        member(C,Biconnected_Group1),
        member(D,Biconnected_Group1),
        not member(A,Biconnected_Group1),!,fail,!.

%If the problem can not be solved with previous cases
%
check_physical_overlap2([C,*,D],[A]):- check_physical_overlap8([C,*,D],[A]).
%=======================================================
% deals with the cases when one of the policy paths a single node
% (first one) and the other is one is list of nodes.Expands list
% and returns overlap if the node is a member of the list.
%
check_physical_overlap3([A],List):-expand(List,List1),
        member(A,List1).

%=======================================================
% deals with the cases when one of the policy paths a single node
% (second one) and the other is one is list of nodes.Expands list
% and returns overlap if the node is a member of the list.
%
check_physical_overlap4(List,[A]):-expand(List,List1),
        member(A,List1).


%=======================================================
% Deals with the cases when both of the policy paths use
% wildcard charachters.
%

% Return overlap if the paths are the same
%
check_physical_overlap5([A,*,B],[C,*,D]):-A=@=C,B=@=D,!.

% If both paths have the same destination, then they
% overlap if both source nodes are in the same biconnected
% component
check_physical_overlap5([A,*,B],[C,*,D]):-
        B=@=D,bic_list(_,Biconnected_Group),
        (member(A,Biconnected_Group), member(C,Biconnected_Group)),!.

% If the source nodes are the same, there is an overlap
% if the destination nodes are in the same biconnected
```

```
% component
%
check_physical_overlap5([A,*,B],[C,*,D]):-
          A=@=C,bic_list(_,Biconnected_Group),
       (member(B,Biconnected_Group), member(D,Biconnected_Group)),!.

% If the source nodes are in the same biconnected component,
% and the destination nodes are together in a different biconnected
% component, returns overlap.
%
check_physical_overlap5([A,*,B],[C,*,D]):-bic_list(_,Biconnected_Group1),
      bic_list(_,Biconnected_Group2),
      member(A,Biconnected_Group1), member(C,Biconnected_Group1),
      member(B,Biconnected_Group2), member(D,Biconnected_Group2),!.

% Deals with the cases when one source-destination pair
% in one biconnected component, and the other is in a
% different one. In this case there is no need to compare
% paths. it returns 'no overlap'.
%
check_physical_overlap5([A,*,B],[C,*,D]):-bic_list(_,Biconnected_Group1),
         member(A,Biconnected_Group1), member(B,Biconnected_Group1),
         not member(C,Biconnected_Group1),
         not member(D,Biconnected_Group1),!,fail,!;
         bic_list(_,Biconnected_Group1),member(C,Biconnected_Group1),
         member(D,Biconnected_Group1),
         not member(A,Biconnected_Group1),
         not member(B,Biconnected_Group1),!,fail,!.

% Deals with the cases when both of the policy paths use
% wildcard characters. deals with the cases if all
% of the nodes in the same biconnected component.Paths
% must be expanded. But this rule Sends the
% biconnected component members to the subroutine to limit
% the number of visited nodes while detecting paths.
%
check_physical_overlap5([A,*,B],[C,*,D]):-bic_list(_,Biconnected_Group),
     (member(A,Biconnected_Group), member(B,Biconnected_Group),
      member(C,Biconnected_Group), member(D,Biconnected_Group)),
      limited_expand_paths(A,B,Biconnected_Group,Path1),!,
      limited_expand_paths(C,D,Biconnected_Group,Path2),!,
      check_physical_overlap(Path1,Path2),!.

%If the problem can not be solved with previous cases
%
check_physical_overlap5([A,*,B],[C,*,D]):-
```

```
        check_physical_overlap8([A,*,B],[C,*,D]).

%=======================================================
% Deals with the cases when one of the policy paths is
% a link and the other uses wildcard characters.
%

% Returns overlap if the link nodes are the same with
% the source and destination nodes
%
check_physical_overlap6([A,B],[C,*,D]):- A=@=C,B=@=D.

% If the link nodes and the source and destination nodes are
% in different biconnected components, there is no need to
% expand path. It returns 'no overlap'.
%
check_physical_overlap6([A,B],[C,*,D]):-bic_list(_,Biconnected_Group1),
        member(C,Biconnected_Group1),member(D,Biconnected_Group1),
        not member(A,Biconnected_Group1),
        not member(B,Biconnected_Group1),!,fail,!.

% If link nodes and the source and destination nodes are
% in the same biconnected component, it calls subroutine
% to expand path again with limited number of nodes.
% After that checks whether they overlap or not.
%
check_physical_overlap6([A,B],[C,*,D]):-bic_list(_,Biconnected_Group),
        (member(A,Biconnected_Group), member(B,Biconnected_Group),
        member(C,Biconnected_Group), member(D,Biconnected_Group)),
        limited_expand_paths(C,D,Biconnected_Group,Path2),!,
        check_physical_overlap(Path1,Path2),!.

%If the problem can not be solved with previous cases
%
check_physical_overlap6([A,B],[C,*,D]):-
        check_physical_overlap8([A,B],[C,*,D]).

%=======================================================
% Deals with the cases when one of the policy paths is
% a link and the other uses wildcard characters.
%

% Returns overlap if the link nodes are the same with
% the source and destination nodes
%
check_physical_overlap7([C,*,D],[A,B]):- A=@=C,B=@=D.
```

```
% If link nodes and the source and destination nodes are
% in the same biconnected component, it calls subroutine
% to expand path again with limited number of nodes.
% After that checks whether they overlap or not.
%
check_physical_overlap7([C,*,D],[A,B]):-bic_list(_,Biconnected_Group),
        (member(A,Biconnected_Group), member(B,Biconnected_Group),
        member(C,Biconnected_Group), member(D,Biconnected_Group)),
        limited_expand_paths(C,D,Biconnected_Group,Path2),!,
        check_physical_overlap(Path2,Path1),!.

% If link nodes and the source and destination nodes are
% in the same biconnected component, it calls subroutine
% to expand path again with limited number of nodes.
% After that checks whether they overlap or not.
%
check_physical_overlap7([C,*,D],[A,B]):-bic_list(_,Biconnected_Group),
        (member(A,Biconnected_Group), member(B,Biconnected_Group),
        member(C,Biconnected_Group), member(D,Biconnected_Group)),
        limited_expand_paths(C,D,Biconnected_Group,Path2),!,
        check_physical_overlap(Path2,Path1),!.

%If the problem can not be solved with previous cases
%
check_physical_overlap7([C,*,D],[A,B]):-
        check_physical_overlap8([C,*,D],[A,B]).

%=======================================================
% deals with the rest of the cases which can not be solved by
% previous rules. Checks is a list is a sublist of the other one.
%
check_physical_overlap8(S,L):-expand(S,S1),expand(L,L1),
        conc(_,L2,L1), conc(S1,_,L2),!.

%=======================================================
% It calls depth limited bidirectional search algorithm
% to find all paths between two nodes. Depth value is fixed
% to 15 from both sides (forward and backward) and can simply
% be changed by changing the values in pharantehesis.
% After finding all paths between two nodes,
% they are recorded to the memory. In case it is required
% to expand same nodes for a different policy definition,
% rule returns already expanded paths without a search
%
expand_paths(A, Z,Expanded_Paths) :-
        setof(Path,bisearch(A, Z, Path, 15, 15),Expanded_Paths),
```

```
        assertz(already_expanded(A,Z,Expanded_Paths)).

%=======================================================
% Depth Limited Bidirectional Search Algorithm
% in depth value is sent from calling rule. If
% the possible paths already detected before,
% it simply returns the list of expanded paths.
%
bisearch(Node1,Node2,Expanded_Paths,_,_):-
        already_expanded(Node1,Node2,Expanded_Paths),!.

bisearch(Node,Node,[Node],_,_).

bisearch(Node1,Node2,[Node1,Node2],_,_) :-
        (link(_,Node1,Node2); link(_,Node2,Node1)).

bisearch(Node1,Node2,AnsPath, Depth1, Depth2) :-
        retractall(agenda(_,_,_,_)),
        retractall(usedstate(_,_,_,_)), retractall(counter(_)),
        retractall(answer(_)),assertz(agenda(Node1,[Node1],front, Depth1)),
        assertz(agenda(Node2,[Node2],back, Depth2)), repeatifagenda,
        ((agenda(State,Path,front, DepthF), agenda(StateB,PathB,back, DepthB),
        add_successors(front,State,Path,DepthF),
        add_successors(back,StateB,PathB,DepthB));
        (agenda(State,Path,front, DepthF),
        not agenda(StateB,PathB,back, DepthB),
        add_successors(front,State,Path,DepthF) );
        (not agenda(State,Path,front, DepthF),
        agenda(StateB,PathB,back, DepthB),
        add_successors(back,StateB,PathB,DepthB))),
        ((agenda(State3,Path3,front, Depth3),
        (agenda(State3,Path4,back, Depth4);
         usedstate(State3,Path4,back, Depth4)));
        (agenda(State3,Path4,back, Depth5),
        (agenda(State3,Path3,front, Depth6);
         usedstate(State3,Path4,front, Depth6)))),
        Path3=[_|XP0], reverse(XP0,XP1), append(XP1,Path4,AnsPath),
         not duplication(AnsPath), not answer(AnsPath),
         assertz(answer(AnsPath)).

bisearch(_,_,_,_) :- fail.
```

```
%=========================================================
% Helps to Depth Limited Bidirectional Search Algorithm
%
repeatifagenda.

repeatifagenda :- agenda(_,_,_,_), !, repeatifagenda.

%=========================================================
% Helps to Depth Limited Bidirectional Search Algorithm
% Adds successors to nodes if the depth limit is not
% exceeded.
%
add_successors(Dir,State,Path,Depth) :-
        Depth >0, NewDepth is Depth-1,
        (link(_,State,Newstate);
         link(_,Newstate,State)), not member(Newstate,Path),
        not agenda(Newstate,[Newstate|Path],Dir, NewDepth),
        not usedstate(Newstate,[Newstate|Path],Dir, NewDepth),
        assertz(agenda(Newstate,[Newstate|Path],Dir, NewDepth)), fail.

add_successors(Dir,State,Path, Depth) :-
        retract(agenda(State,Path,Dir, Depth)),
        assertz(usedstate(State,Path,Dir, Depth)).

%=========================================================
% Helps to Depth Limited Bidirectional Search Algorithm
% Checkes if the list have duplicate members
%
duplication([X|L]) :- member(X,L), !.

duplication([_|L]) :- duplication(L).

%=========================================================
% It calls revised version of the Depth Limited
% Bidirectional Search Algorithm to fing the paths
% between two nodes.
%
limited_expand_paths(A, Z,Biconnected_Components,Expanded_Paths) :-
        setof(Path,limited_bisearch(A, Z, Path, 15, 15,Biconnected_Components),
        Expanded_Paths),assertz(already_expanded(A,Z,Expanded_Paths)).

%=========================================================
% Revised version of the Depth Limited Bidirectional
% Search Algorithm. It only uses the nodes belong
% to one biconnected component. It helps to
% limit the number of nodes visited by decreasing
```

```
% the search space. Again the paths expandd before
% returned without a search.
%
limited_bisearch(Node1,Node2,Expanded_Paths,_,_,_):-
        already_expanded(Node1,Node2,Expanded_Paths),!.

limited_bisearch(Node,Node,[Node],_,_,_).

limited_bisearch(Node1,Node2,[Node1,Node2],_,_,_) :-
        (link(_,Node1,Node2); link(_,Node2,Node1)).

limited_bisearch(Node1,Node2,AnsPath, Depth1, Depth2,
        Biconnected_Components) :-
        retractall(agenda(_,_,_,_)),retractall(usedstate(_,_,_,_)),
        retractall(counter(_)), retractall(answer(_)),
        assertz(agenda(Node1,[Node1],front, Depth1)),
        assertz(agenda(Node2,[Node2],back, Depth2)), repeatifagenda,
        ((agenda(State,Path,front, DepthF), agenda(StateB,PathB,back, DepthB),
        add_successors1(front,State,Path,DepthF,Biconnected_Components),
        add_successors1(back,StateB,PathB,DepthB,Biconnected_Components) );
        (agenda(State,Path,front, DepthF),
         not agenda(StateB,PathB,back, DepthB),
        add_successors1(front,State,Path,DepthF,Biconnected_Components) );
        (not agenda(State,Path,front, DepthF),
         agenda(StateB,PathB,back, DepthB),
        add_successors1(back,StateB,PathB,DepthB,Biconnected_Components))),
        ((agenda(State3,Path3,front, Depth3),
        (agenda(State3,Path4,back, Depth4);
         usedstate(State3,Path4,back, Depth4)));
        (agenda(State3,Path4,back, Depth5),
        (agenda(State3,Path3,front, Depth6);
         usedstate(State3,Path4,front, Depth6)))),
        Path3=[_|XP0], reverse(XP0,XP1), append(XP1,Path4,AnsPath),
        not duplication(AnsPath), not answer(AnsPath),
        assertz(answer(AnsPath)).

limited_bisearch(_,_,_,_,_) :- fail.

%=======================================================
% Helps to Depth Limited Bidirectional Search Algorithm
% Adds successors to nodes from the biconnected
% component list if the depth limit is not exceeded.
%
add_successors1(Dir,State,Path,Depth,Biconnected_Components) :-
        Depth >0, NewDepth is Depth-1,
        (member(Newstate,Biconnected_Components),
```

```
        link(_,State,Newstate); link(_,Newstate,State)),
        not member(Newstate,Path),not agenda(Newstate,[Newstate|Path],
        Dir, NewDepth),
        not usedstate(Newstate,[Newstate|Path],Dir, NewDepth),
        assertz(agenda(Newstate,[Newstate|Path],Dir, NewDepth)), fail.

add_successors1(Dir,State,Path, Depth,Biconnected_Components) :-
        retract(agenda(State,Path,Dir, Depth)),
        assertz(usedstate(State,Path,Dir, Depth)).

%=======================================================
% print_conflict_list:
% Outputs nicely each policy conflict that could not
% be resolved using the ID of policy's creator
%
print_unresolved_conflict_list([]).

print_unresolved_conflict_list([[_,_,_,_,_,_,[]]|Tail]):-
        print_unresolved_conflict_list(Tail).

print_unresolved_conflict_list([[Policy1,Path1,Target1,Policy2,Path2,
        Target2,Permit_results]|Tail]) :-
        policy_owner(Policy1,Owner1),
        policy_owner(Policy2,Owner2),
        user(Owner1,Value1),
        user(Owner2,Value2),
        Value1 = Value2,
        nl,nl,write('Conflict '),
        write(Policy1),
        write(' <==> '),
        write(Policy2),nl,
        write('   '),
        write(Policy1),
        write(' Path = ['),
        write_path(Path1),nl,
        write('   '),
        write(Policy1),
        write(' Targets: '),
        write_targets(Target1),nl,
        write('   '),
        write(Policy2),
        write(' Path = ['),
        write_path(Path2),nl,
        write('   '),
        write(Policy2),
        write(' Targets: '),
```

```
        write_targets(Target2),nl,
        write('   '),
        write('Target Conflicts: '),
        print_list(Permit_results),nl,
        print_unresolved_conflict_list(Tail).

print_unresolved_conflict_list([[Policy1,_,_,Policy2,_,_,_]|Tail]) :-
        policy_owner(Policy1,Owner1),
        policy_owner(Policy2,Owner2),
        user(Owner1,Value1),
        user(Owner2,Value2),
        not(Value1 = Value2),
        print_unresolved_conflict_list(Tail).

%=========================================================
% print_conflict_list:
% Outputs nicely policy conflicts that could be
% resolved using the ID of the policy's creator
%
print_resolved_conflict_list([]).

print_resolved_conflict_list([[_,_,_,_,_,_,[]]|Tail]):-
        print_resolved_conflict_list(Tail).

print_resolved_conflict_list([[Policy1,Path1,Target1,Policy2,Path2,
        Target2,Permit_results]|Tail]) :-
        policy_owner(Policy1,Owner1),
        policy_owner(Policy2,Owner2),
        user(Owner1,Value1),
        user(Owner2,Value2),
        not(Value1 = Value2),
        nl,nl,write('Conflict '),
        write(Policy1),
        write(' <==> '),
        write(Policy2),nl,
        write('   '),
        write(Policy1),
        write(' Path = ['),
        write_path(Path1),nl,
        write('   '),
        write(Policy1),
        write(' Targets: '),
        write_targets(Target1),nl,
        write('   '),
        write(Policy2),
        write(' Path = ['),
```

```
        write_path(Path2),nl,
        write('   '),
        write(Policy2),
        write(' Targets: '),
        write_targets(Target2),nl,
        write('   '),
        write('Target Conflicts: '),
        print_list(Permit_results),nl,
        print_how_resolved(Policy1,Value1,Policy2,Value2),
        print_resolved_conflict_list(Tail).

print_resolved_conflict_list([[Policy1,_,_,Policy2,_,_,_]|Tail]) :-
        policy_owner(Policy1,Owner1),
        policy_owner(Policy2,Owner2),
        user(Owner1,Value1),
        user(Owner2,Value2),
        Value1 = Value2,
        print_resolved_conflict_list(Tail).

%=======================================================
% Prints nicely the how the a policy conflict was resolved
%
print_how_resolved(Policy1,Value1,Policy2,Value2):-
        Value1 < Value2,
        write('   '),
        write('Resolved: '),
        write(Policy1),
        write('(Priority = '),
        write(Value1),
        write(')'),
        write(' overrides=> '),
        write(Policy2),
        write('(Priority = '),
        write(Value2),
        write(') '),nl.

print_how_resolved(Policy1,Value1,Policy2,Value2):-
        Value2 < Value1,
        write('   '),
        write('Resolved: '),
        write(Policy2),
        write('(Priority = '),
        write(Value2),
        write(')'),
        write(' overrides=> '),
        write(Policy1),
```

```
        write('(Priority = '),
        write(Value1),
        write(') '),nl.

%============================================================
% Prints nicely a list of elements
%
print_list([]).

print_list([[C|V] | Tail]):-
        not(empty(Tail)),
        write(C),
        write(' = '),
        write(V),
        write(', '),
        print_list(Tail).

print_list([[C|V] | Tail]):-
        empty(Tail),
        write(C),
        write(' = '),
        write(V).

%============================================================
% write_path:
% is a helper function used to output
% a list with all the atoms quoted.

write_path([]):- write(']').

write_path([X|[]]):-
                write(X),
                write(']'),!,true.

write_path([X|Tail]):-
                 write(X),
                 write(','),
                 write_path(Tail).

%============================================================
% writes nicely the target list of a policy
%
write_targets([]):- write('[]').

write_targets([A,C,_,V|[]]):-
        write(A),
```

```
        write(' '),
        write(C),
        write('=['),
        write(V),
        write(']'),!,true.

write_targets([A,C,_,V|Tail]):-
        write(A),
        write(' '),
        write(C),
        write('=['),
        write(V),
        write('], '),
        write_targets(Tail).

%=========================================================
% Prints out any message conflicts beween a path
% and the links used to compose it.
% First step is to generate the list of conflicts
%       step 2 is to output a list if not empty
%
print_message_conflict_list:-
        not(setof(Link_Message,
         print_message_conflict_list_helper(Link_Message),_)),fail.

print_message_conflict_list:-
        setof(Link_Message,
         print_message_conflict_list_helper(Link_Message),Message_Conflicts),
        write_message_conflicts(Message_Conflicts),fail.

%=========================================================
% Returns a list of links and nodes that do not support
% the "messages" require by a policy path
%
print_message_conflict_list_helper(Message_Conflicts):-
        path(Policy,_,_),setof(Link_Message,
        message_conflict(Policy,Link_Message),Message_Conflicts).

%=========================================================
% Prints nicely all the policies that contain message conflicts
%
write_message_conflicts([]).

write_message_conflicts([Message_Conflict|Tail]):-
        write_message_conflict(Message_Conflict),nl,
        write_message_conflicts(Tail).
```

```
%=========================================================
% Writes out a individual message conflict for a policy
%
write_message_conflict([]).

write_message_conflict([[Policy, Message_list] | Policy_Message_Tail]) :-
        write(Policy),write(' requires message support on the following links'),nl,
        write_message_conflict_helper1(Message_list),
        write_message_conflict(Policy_Message_Tail).

%=========================================================
% Prints each link that does not support the "message" on the path
%
write_message_conflict_helper1([]).

write_message_conflict_helper1([[Link,Message] |Tail]):-
        write('     '),
        write('link '''),write(Link),write(''' requires support
        for message '''),write(Message),write(''''),nl,
        write_message_conflict_helper1(Tail).

%=========================================================
% Expand Policy paths. if the path is using wildcard character,
% it uses Depth Limited Bidirectional Search or its revised version.
% if the path is a list of nodes created by user, it checks whether
% that path really exists in network or not.
%
expand([A,'*',B|Tail],Path):- bic_list(_,Biconnected_Components),
        (member(A,Biconnected_Components),
        member(B,Biconnected_Components)),
        limited_expand_paths(A,B,Biconnected_Components,Path1),
        expand(Tail,Expanded_Tail), Path2=[Path1|Expanded_Tail],
        flatten(Path2,Path).

expand([A,'*',B|Tail],Path):-
        expand_paths(A,B,Path1),
        expand(Tail,Expanded_Tail), Path2=[Path1|Expanded_Tail],
        flatten(Path2,Path).

expand([X],[X]).

expand([X,Y|Tail],[X,Y|Path]):-not(X=@='*'),(link(_,X,Y);link(_,Y,X)),
        Tail1=[Y|Tail], expand(Tail1, Path).

expand([],[]).
```

```
%=========================================================
% Helper Prolog rule representin If-Then statements
%
ifthen(P,Q):-call(P),!,call(Q).

ifthen(_,_).


%=========================================================
%  Removes duplicate path definitions from the memory to prevent redundant
% path comprasions.
%
clean_memory:-path(Label1,Path1,Target1),
        path(Label2,Path2,Target2),
        ifthen(path(Label1,Path1,Target1)=@=path(Label2,Path2,Target2),
        retract(path(Label2,Path2,Target2))).
```

APPENDIX E. IMPORTANT ARTICULATION POINT THEOREMS

(Source: Prof. Geoffrey Xie)

A. THEOREM-1

Consider a fully connected topology with bidirectional links. Let a and b be two arbitrary nodes in the topology. If the topology does not have any articulation point, then for any link in the topology, there exists a loop-free path in $< a,*,b >$ that goes through the link.

B. THEOREM-2

Consider a fully connected topology with bidirectional links. Let a and b be two distinct nodes and $c \mapsto d$ an outbound interface in the topology. No path in $< a,*,b >$ goes through interface $c \mapsto d$ if and only if removal of all nodes and links of any path in $< a,*,c >$ will disconnect nodes d and b.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. THE COMPILING RESULT FILE OF THE 40-NODE SAMPLE FILE WITH THE PREVIOUS COMPILER

(The previous compiler can not detect all conflicts)

Print Unresolved Conflicts

==========

Conflict Policy7 <==> Policy13

Policy7 Path = [n17]

Policy7 Targets: []

Policy13 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy13 Targets: permit traffic_class=[data]

Target Conflicts: traffic_class = [data]

Conflict Policy7 <==> Policy15

Policy7 Path = [n17]

Policy7 Targets: []

Policy15 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy15 Targets: permit traffic_type=[research]

Target Conflicts: traffic_type = [research]

Conflict Policy7 <==> Policy20

Policy7 Path = [n17]

Policy7 Targets: []

Policy20 Path = [n19,n10,n22,n17,n8,n16]

Policy20 Targets: permit traffic_security=[Private]

Target Conflicts: traffic_security = [Private]

Conflict Policy7 <==> Policy4

Policy7 Path = [n17]

Policy7 Targets: []

Policy4 Path = [n19,n10,n22,n17,n8,n16]

Policy4 Targets: []

Target Conflicts: Policy7 = [deny_all]

Print Resolved Conflicts

==========

Conflict Policy7 <==> Policy8

Policy7 Path = [n17]

Policy7 Targets: []

Policy8 Path = [n5,n3,n17,n22,n30,n28]

Policy8 Targets: []

Target Conflicts: Policy7 = [deny_all]

Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

APPENDIX G. THE COMPILING RESULT FILE OF THE 40-NODE SAMPLE FILE WITH THE IMPROVED COMPILER

(The improved compiler can detect conflicts that could not be detected by the previous version. The additional conflicts are marked using an italic font.)

Print Unresolved Conflicts

==========

Conflict Policy13 <==> Policy15

Policy13 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy13 Targets: permit traffic_class=[data]

Policy15 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy15 Targets: permit traffic_type=[research]

Target Conflicts: traffic_type = [research]

Conflict Policy13 <==> Policy7

Policy13 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy13 Targets: permit traffic_class=[data]

Policy7 Path = [n17]

Policy7 Targets: []

Target Conflicts: Policy7 = [deny_all]

Conflict Policy15 <==> Policy13

Policy15 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy15 Targets: permit traffic_type=[research]

Policy13 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy13 Targets: permit traffic_class=[data]

Target Conflicts: traffic_class = [data]

Conflict Policy15 <==> Policy7

Policy15 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy15 Targets: permit traffic_type=[research]

Policy7 Path = [n17]

Policy7 Targets: []

Target Conflicts: Policy7 = [deny_all]

Conflict Policy17 <==> Policy8

Policy17 Path = [n1,n13,n8]

Policy17 Targets: deny node_traffic=[n4], deny node_traffic=[n8]

Policy8 Path = [n1,n13,n8]

Policy8 Targets: []

Target Conflicts: Policy8 = [permit_all]

Conflict Policy19 <==> Policy20

Policy19 Path = [n19,n10,n22,n17,n8,n16]

Policy19 Targets: []

Policy20 Path = [n19,n10,n22,n17,n8,n16]

Policy20 Targets: permit traffic_security=[Private]

Target Conflicts: traffic_security = [Private]

Conflict Policy19 <==> Policy4

Policy19 Path = [n19,n10,n22,n17,n8,n16]

Policy19 Targets: []

Policy4 Path = [n19,n10,n22,n17,n8,n16]

Policy4 Targets: []

Target Conflicts: Policy19 = [deny_all]

Conflict Policy20 <==> Policy19

Policy20 Path = [n19,n10,n22,n17,n8,n16]

Policy20 Targets: permit traffic_security=[Private]

Policy19 Path = [n19,n10,n22,n17,n8,n16]

Policy19 Targets: []

Target Conflicts: Policy19 = [deny_all]

Conflict Policy20 <==> Policy7

Policy20 Path = [n19,n10,n22,n17,n8,n16]

Policy20 Targets: permit traffic_security=[Private]

Policy7 Path = [n17]

Policy7 Targets: []

Target Conflicts: Policy7 = [deny_all]

Conflict Policy4 <==> Policy19

Policy4 Path = [n19,n10,n22,n17,n8,n16]

Policy4 Targets: []

Policy19 Path = [n19,n10,n22,n17,n8,n16]

Policy19 Targets: []

Target Conflicts: Policy4 = [permit_all]

Conflict Policy4 <==> Policy20

Policy4 Path = [n19,n10,n22,n17,n8,n16]

Policy4 Targets: []

Policy20 Path = [n19,n10,n22,n17,n8,n16]

Policy20 Targets: permit traffic_security=[Private]

Target Conflicts: traffic_security = [Private]

Conflict Policy4 <==> Policy7

Policy4 Path = [n19,n10,n22,n17,n8,n16]

Policy4 Targets: []

Policy7 Path = [n17]

Policy7 Targets: []

Target Conflicts: Policy4 = [permit_all]

Conflict Policy7 <==> Policy13

Policy7 Path = [n17]

Policy7 Targets: []

Policy13 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy13 Targets: permit traffic_class=[data]

Target Conflicts: traffic_class = [data]

Conflict Policy7 <==> Policy15

Policy7 Path = [n17]

Policy7 Targets: []

Policy15 Path = [n24,n30,n22,n17,n8,n13,n2]

Policy15 Targets: permit traffic_type=[research]

Target Conflicts: traffic_type = [research]

Conflict Policy7 <==> Policy20

Policy7 Path = [n17]

Policy7 Targets: []

Policy20 Path = [n19,n10,n22,n17,n8,n16]

Policy20 Targets: permit traffic_security=[Private]

Target Conflicts: traffic_security = [Private]

Conflict Policy7 <==> Policy4

Policy7 Path = [n17]

Policy7 Targets: []

Policy4 Path = [n19,n10,n22,n17,n8,n16]

Policy4 Targets: []

Target Conflicts: Policy7 = [deny_all]

Conflict Policy8 <==> Policy17

Policy8 Path = [n1,n13,n8]

Policy8 Targets: []

Policy17 Path = [n1,n13,n8]

Policy17 Targets: deny node_traffic=[n4], deny node_traffic=[n8]

Target Conflicts: Policy8 = [permit_all]

Print Resolved Conflicts

==========

Conflict Policy2 <==> Policy8

Policy2 Path = [n5,n3,n17,n22,n30,n28]

Policy2 Targets: deny node_traffic=[n11]

Policy8 Path = [n5,n3,n17,n22,n30,n28]

Policy8 Targets: []

Target Conflicts: Policy8 = [permit_all]

Resolved: Policy8(Priority = 3) overrides=> Policy2(Priority = 4)

Conflict Policy7 <==> Policy8

Policy7 Path = [n17]

Policy7 Targets: []

Policy8 Path = [n5,n3,n17,n22,n30,n28]

Policy8 Targets: []

Target Conflicts: Policy7 = [deny_all]

Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

Conflict Policy8 <==> Policy2

Policy8 Path = [n5,n3,n17,n22,n30,n28]

Policy8 Targets: []

Policy2 Path = [n5,n3,n17,n22,n30,n28]

Policy2 Targets: deny node_traffic=[n11]

Target Conflicts: Policy8 = [permit_all]

Resolved: Policy8(Priority = 3) overrides=> Policy2(Priority = 4)

Conflict Policy8 <==> Policy7

Policy8 Path = [n5,n3,n17,n22,n30,n28]

Policy8 Targets: []

Policy7 Path = [n17]

Policy7 Targets: []

Target Conflicts: Policy8 = [permit_all]

Resolved: Policy7(Priority = 1) overrides=> Policy8(Priority = 3)

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX H. 80-NODE MESH TOPOLOGY SAMPLE FILE USED IN TESTING MORE DIFFUCULT CASES

define node n1,n2,n3,n4,n5,n6,n7,n8,n9,n10;
define node n11,n12,n13,n14,n15,n16,n17,n18,n19,n20;
define node n21,n22,n23,n24,n25,n26,n27,n28,n29,n30;
define node n31,n32,n33,n34,n35,n36,n37,n38,n39,n40;
define node n41,n42,n43,n44,n45,n46,n47,n48,n49,n50;
define node n51,n52,n53,n54,n55,n56,n57,n58,n59,n60;
define node n61,n62,n63,n64,n65,n66,n67,n68,n69,n70;
define node n71,n72,n73,n74,n75,n76,n77,n78,n79,n80;

define link n13_n1 <n13,n1>,n13_n2 <n13,n2>, n13_n12 <n13,n12>;
define link n3_n9 <n3,n9>, n3_n4 <n3,n4>, n3_n5 <n3,n5>, n3_n14 <n3,n14>;
define link n7_n6 <n7,n6>, n6_n11 <n6,n11>,n11_n17 <n11,n17>, n11_n18 <n11,n18>;
define link n8_n13 <n8,n13>, n8_n16 <n8,n16>, n8_n20 <n8,n20>, n8_n21 <n8,n21>;
define link n17_n8 <n17,n8>, n17_n3 <n17,n3>, n17_n22 <n17,n22>;
define link n22_n10 <n22,n10>, n22_n25 <n22,n25>, n22_n30 <n22,n30>;
define link n10_n15 <n10,n15>, n10_n19 <n10,n19>;
define link n25_n23 <n25,n23>, n23_n35 <n23,n35>, n35_n27 <n35,n27>;
define link n30_n24 <n30,n24>, n30_n29 <n30,n29>, n30_n28 <n30,n28>, n30_n33 <n30,n33>;
define link n26_n30 <n26,n30>, n26_n31 <n26,n31>,n26_n7 <n26,n7>;
define link n33_n32 <n33,n32>;
define link n32_n36 <n32,n36>, n32_n37 <n32,n37>, n32_n38 <n32,n38>;
define link n31_n38 <n31,n38>, n31_n39 <n31,n39>, n31_n40 <n31,n40>, n31_n34 <n31,n34>;
define link n11_n41 <n11,n41>;
define link n12_n42 <n12,n42>;
define link n41_n53 <n41,n53>, n42_n53 <n42,n53>, n52_n53 <n52,n53>;
define link n53_n48 <n53,n48>, n48_n56 <n48,n56>, n48_n60 <n48,n60>, n48_n61 <n48,n61>;
define link n48_n57 <n48,n57>, n57_n43 <n57,n43>, n43_n49 <n43,n49>;
define link n43_n44 <n43,n44>, n45_n43 <n45,n43>, n43_n54 <n43,n54>;
define link n57_n51 <n57,n51>, n51_n46 <n51,n46>, n46_n47 <n46,n47>;
define link n51_n58 <n51,n58>, n58_n62 <n58,n62>, n57_n62 <n57,n62>;
define link n62_n50 <n62,n50>, n50_n55 <n50,n55>, n50_n59 <n50,n59>;
define link n62_n70 <n62,n70>, n70_n64 <n70,n64>, n70_n69 <n70,n69>;
define link n70_n68 <n70,n68>, n70_n73 <n70,n73>, n70_n66 <n70,n66>;
define link n73_n72 <n73,n72>, n72_n76 <n72,n76>, n72_n77 <n72,n77>;
define link n72_n78 <n72,n78>, n78_n71 <n78,n71>, n66_n71 <n66,n71>;
define link n71_n79 <n71,n79>, n71_n80 <n71,n80>, n71_n74 <n71,n74>;
define link n62_n65 <n62,n65>, n65_n63 <n65,n63>, n63_n75 <n63,n75>;

```
define link n75_n67 <n75,n67>,n75_n1 <n75,n1>;

/*
 * Define the paths used in the network policies
 */
define path n27_n25 {<n27,*, n25>};
define path n27_n23 {<n27, *, n23>};
define path n1_n8 {<n1,*,n8>};
define path n12_n17 {<n12, *, n17>};
define path n7_n37 {<n7,*,n37>};
define path n19_n16 {<n19,*,n16>};
define path n5_n28 {<n5,*,n28>};
define path n24_n2 {<n24,*,n2>};
define path n30_n70 {<n30,*,n70>};
define path n66_n77 {<n66,*,n77>};
define path n19_n55 {<n19,*,n55>};
define path n71_n44 {<n71,*,n44>};
define path n11_n39 {<n11,*,n39>};
define path n39_n27 {<n39,n31,n26,n30,n22,n25,n23,n35,n27>};

/*
 * Define the users who can create policies
 */
define policy_maker Net_Manager(1), Xie(3), Rowe(3), Stone(4);

/*
 * User defined classes of traffic which can be used
 * in the target element of policy rules
 */
define class traffic_type {research, university};
define class traffic_class {data, video, voice};
define class traffic_security {Private, Public};
define class traffic_priority {High, Med, Low};
define class node_traffic {n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12};

/*
 * User defined type that can be used in the conditional element
 * of a policy rule.
 */
define type day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday};

/*
 * Paramaters associated with links and paths
 */
define link_param n13_n1 {BW := 100MBPS, delay(), loss_rate(),jitter()};
```

```
/*
 * Format of Policy Term:
 * PolicyID UserID {paths} {target} {conditions} {action_items};
 */


Policy1 Net_Manager {n27_n25,n30_n70} {*}{jitter() < 50 MSEC, delay() < 10
Msec}{permit};

/*
 * Deny all traffic from NSF on the link between NASA and SPAWAR
 */
Policy2 Stone {n5_n28,n7_n37} {node_traffic == {n11}} {*} {deny};

/*
 * Permit all traffic unconditionally on the two paths
 * specified with the NPS_CERT path definition
 */
Policy3 Rowe {n7_n37,n66_n77} {*} {*} {permit};

/*
 * Permit all traffic unconditionally on the NASA->IETF link
 */
Policy4 Net_Manager {n19_n16,n19_n55} {*} {*} {permit};

/*
 * Deny all university traffic after 1100 on the
 * NPS->DARPA link
 */
Policy5 Net_Manager {n12_n17,n71_n44} {traffic_type == {university}} {time
>= 1100} {deny};

/*
 * Deny university traffic after 1300 on the
 * NPS->DARPA link
 */
Policy6 Net_Manager {n12_n17,n11_n39} {traffic_type == {university}} {time
>= 1300} {deny};

/*
 * Deny all traffic from hosts with and address beginning
 * with 131.
 */
```

```
Policy7 Net_Manager {n17} {*} {hostIP == 131.*.*.*} {deny};

/*
 * Permit all traffic from hosts whos addresses begin with
 * either 153.20.8 or 131.40.
 */
Policy8 Xie {n1_n8,n5_n28} {*} {hostIP == 153.20.8.*, hostIP == 131.40.*.*}
{permit};

/*
 * Permit and assign a priority of 3 to video traffic before the hour
 * of 0800, on the NPS->DARPA link
 */
Policy9 Net_Manager  {n7_n37} {traffic_class == {video}} {time <=0759}
{priority := 3};

/*
 * Deny video traffic on the NPS->DARPA link between the hours of 0800 and
1600
 */
Policy10 Net_Manager {n7_n37} {traffic_class == {video}} {time >=0800, time
<=1600} {deny};

/*
 * Permit and assign a priority of 5 to video traffic after the hour
 * of 1600 on the NPS->DARPA link
 */
Policy11 Net_Manager {n7_n37} {traffic_class == {video}} {time >=1601}
{priority := 5};

Policy12 Net_Manager {n27_n25,n27_n23} {*}{jitter() < 50 MSEC, loss_rate()
< 20%}{permit};

Policy13 Net_Manager {n24_n2} {traffic_class == {data}} {*} {permit};

Policy14 Net_Manager {n39_n27} {*} {time >= 1600:30, day == Monday}
{deny};

Policy15 Net_Manager {n24_n2,n39_n27,n7_n37} {traffic_type == {research}}
{time >= 0800, time <= 1600} {priority :=1};

Policy16 Stone {n7_n37}{traffic_priority =={Low}} {time >= 0800, time <=
1600} {deny};

Policy17 Xie {n1_n8} {node_traffic == {n4,n8}} {*} {deny};
```

Policy18 Rowe {n7_n37} {traffic_security == {Public}} {*} {deny};

Policy19 Net_Manager {n19_n16} {*} {time >= 0800, time <= 0815} {deny};

Policy20 Net_Manager {n19_n16} {traffic_security == {Private}} {time >=0800, time <= 0815} {permit};

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] C. GOH, A Generic Approach to Policy Description in System Management, HP Labs Technical Reports, July 1996

[2] J. STRASSNER, E. ELLESSON, Terminology for Describing Network Policy and Services, Internet Engineering Task Force, Internet Draft draft-strasner-policy-terms-01.txt, 1998.

[3] M. SLOMAN, Policy Driven Management for Distributed Systems, *Journal of Network and System Management, vol. 2 part 4*, 1994

[4] D. MARRIOTT, M. SLOMAN, Management Policy Service for Distributed Systems, IEEE 3rd International Workshop on Services in Distributed and Networked Environments, 1996.

[5] N. GUERROUDJI, A Management Policies Framework, 2nd IEEE International Workshop on System Management, June 1996

[6] J. D. MOFFECT, Network and Distributed System Management, *IEEE Journal on Selected Areas in Communications, pages 1404–1414, Special Issue on Network Management*, December 1993.

[7] T. KOCH, B. KRAMER, G. ROHDE, A Rule Based Management Architecture, Proceedings of 2nd International Workshop on Services in Distributed Network Enviroment, June1995

[8] S. STEINKE, *Policy Based Networking, Network Magazine*, March 1998

[9] C. WALKER, *Policy Based Networking, Computerworld Magazine*, February 1999

[10] G. N. STONE, A Path-Based Network Policy Language, Computer Science Department, Naval Postgraduate School, Monterey, September 2000.

[11] BRNA, Paul Prolog Programming-A first course. http://www.cbl.leeds.ac.uk/~paul/Prologbook/, September 2002

[12] CAMPBELL, J. A. *Implementations of Prolog*, Ellis Horwood Limited Publishing, 1984

[13] F. KLUZNIAK, S. SZPAKOWICZ, *Prolog for Programmers*, Acedemic Press Inc. 1985

[14] P. MEREL http://c2.com/cgi/wiki?PrologLanguage, November 2002

[15] S. GARAVAGLIA, *Prolog Programming Techniques and Applications*, Harper & Row Publishers, 1987

[16] E. SHAPIRO, *Concurrent Prolog*, The MIT Press, 1987

[17] E. TICK, *Memory Performance of Prolog Architectures*, Kluwer Academic Publishers, 1988

[18] R. SEDGEWICK, *Algorithms*, Addison Wesley Pub., 1992

[19] J. BOYLE, R. COHEN, D. DURHAM, S. HERZOG, R. RAJA, and A. SASTRY, "The COPS (Common Open Policy Service) Protocol", Internet Engineering Task Force, Internet Draft draft-ietf-rap-cops-05.txt, December 1998.

[20] J. CASE, M. FEDOR, M. SCHOFFSTALL, J. DAVIN, "A Simple Network Management Protocol (SNMP)," Internet Engineering Task Force: Network Working Group Request for Comments: 1157, May 1990.

[21] http://www.icsuci.edu/~dan/class/161/notes/8/Bicomps.html , February 2003

[22] http://roseanne.tesoriero.washcoll.edu/ Courses/270/biconnected.doc , February 2003

[23] M. R. GENESERETH, M. L. GINSBERG, *Logic Programming*, Computer Science Dept., Stanford University. Stanford, CA, September 1985

[24] C. MARCUS, *Prolog Programming*, Addison Wesley Publishing Company, 1986

[25] R. K. APT, *The Logic Programming Paradigm and Prolog, Rep. TR-87-35*, Dept of Computer Science, Univ. of Texas at Austin, July 2001

[26] N. ROWE, *Artificial Intelligence Course Notes*, Naval Postgraduate School, Monterey, CA, February 2003

[27] http://www.cse.unsw.edu.au/~billw/cs9414/notes/kr/rules/rules.html, February 2003

[28] M. A. COVINGTON, D. NUTE, A. VELLINO, *Prolog Programming In Depth*, Scott, Foresman and Company Computer Books, 1988

[29] G. G. XIE, D. HENSGEN, T. KIDD, and J. YARGER, SAAM: An Integrated Network Architecture for Integrated Services, presented at Proceedings of 6th IEEE/IFIP International Workshop on Quality of Service, Napa, CA, 1998

[30] T. H. CORMEN, C. E. LEISERSON, R.L. RIVEST, *Introduction to Algorithms*, The MIT Press, 1990

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia 22060

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California 93943

3. Professor Geoffrey Xie
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943

4. Professor Neil Rowe
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943

5. Deniz Kuvvetleri Komutanligi
   Kutuphane
   Bakanliklar, Ankara, TURKEY

6. Deniz Harp Okulu Komutanligi
   Kutuphane
   Tuzla, Istanbul, TURKEY

7. Bilkent Universitesi Kutuphanesi
   Bilkent, Ankara, TURKEY

8. Orta Dogu Teknik Universitesi Kutuphanesi
   Balgat, Ankara, TURKEY

9. Bogazici Universitesi Kutuphanesi
   Bebek, Istanbul, TURKEY

10. Dz. Utgm. Ahmet Guven
    Deniz Kuvvetleri Komutanligi
    Bakanliklar, Ankara, TURKEY